Resource Management of Replicated Service Systems Provisioned in the Cloud

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Mathias Björkqvist

under the supervision of
Prof. Walter Binder

February 2015

Dissertation Committee

Prof. Rolf Krause	Università della Svizzera Italiana, Switzerland
Prof. Cesare Pautasso	Università della Svizzera Italiana, Switzerland
Prof. Heiko Schuldt	University of Basel, Switzerland
Prof. Giuseppe Serazzi	Politechnico di Milano, Italy

Dissertation accepted on 11 February 2015

Research Advisor	PhD Program Director	PhD Program Director
Prof. Walter Binder	**Prof. Igor Pivkin**	**Prof. Stefan Wolf**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Mathias Björkqvist
Lugano, 11 February 2015

Abstract

Service providers seek scalable and cost-effective cloud solutions for hosting their applications. Despite significant recent advances facilitating the deployment and management of services on cloud platforms, a number of challenges still remain. Service providers are confronted with time-varying requests for the provided applications, inter-dependencies between different components, performance variability of the procured virtual resources, and cost structures that differ from conventional data centers. Moreover, fulfilling service level agreements, such as the throughput and response time percentiles, becomes of paramount importance for ensuring business advantages.

In this thesis, we explore service provisioning in clouds from multiple points of view. The aim is to best provide service replicas in the form of VMs to various service applications, such that their tail throughput and tail response times, as well as resource utilization, meet the service level agreements in the most cost effective manner. In particular, we develop models, algorithms and replication strategies that consider multi-tier composed services provisioned in clouds. We also investigate how a service provider can opportunistically take advantage of observed performance variability in the cloud. Finally, we provide means of guaranteeing tail throughput and response times in the face of performance variability of VMs, using Markov chain modeling and large deviation theory. We employ methods from analytical modeling, event-driven simulations and experiments. Overall, this thesis provides not only a multi-faceted approach to exploring several crucial aspects of hosting services in clouds, i.e., cost, tail throughput, and tail response times, but our proposed resource management strategies are also rigorously validated via trace-driven simulation and extensive experiments.

Contents

Chapter 1

Introduction

Providers of service-oriented systems aim at delivering satisfactory performance in a cost-effective manner, which today often means taking advantage of the cloud computing paradigm in one way or another. On the one hand, the operational cost is proportional to the number resources deployed, such as physical machines or *virtual machines* (VMs) hosting service replicas. For fast-growing enterprises, or for services experiencing large time variability, provisioning some or all of these resources in clouds can provide significant benefits in terms of ease-of-management or cost. On the other hand, system performance, e.g., response time and resource utilization, hinges on the capability of the provisioned resources in processing time-varying requests and the balancing of the load across replicas. Related studies [Zhang et al., 2008; Singh et al., 2010] show that striking a good balance between conflicting objectives, i.e., operational cost and performance, is not an easy task, especially in multi-tier systems. Statically providing a maximum number of resources may guarantee the performance at a high operational cost, whereas unbalanced loads and under-provisioned resources could lead to a significant performance degradation. Dynamically and accurately adjusting service capacities, i.e., the number and size of physical or virtual machines, depending on the workload, has been shown to be effective in solving the dilemma of balancing between performance targets and operational cost. The ease of dynamically adjusting resources is one of the key advantages of clouds, which is why more and more service providers are turning to them for their service provisioning needs [Chen et al., 2005; Lin et al., 2013].

The rest of this chapter gives an overview of the key relevant service provisioning aspects from the perspective of service providers and the cloud computing paradigm. Section 1.1 gives a general description of service systems and the cloud computing paradigm. Provisioning of replicated, multi-tier and composed

services in clouds is presented in Section 1.2. The public cloud, and how service providers can make use of it, is introduced in Section 1.3. This section also touches upon two of the key properties of public clouds, namely the performance variation, and their cost structures from a service provider's perspective. Load balancing of replicated and distributed systems is presented in Section 1.4. In Section 1.5, the problem statement is introduced, followed by an overview of the rest of the thesis in Section 1.6 and the contributions in Section 1.7.

1.1 Overview of Service Systems and Cloud Computing

1.1.1 Service Systems

Service-oriented systems are commonly composed of distributed web services [Alonso et al., 2004; Papazoglou et al., 2008]. Applications' requests, consisting of multiple invocations of web services, show a strong time varying behavior, e.g., time of day and day of the week effects [Arlitt and Jin, 2000; Chen et al., 2005; Singh et al., 2010; Stewart et al., 2007]. Such systems process requests either as atomic services, or by invoking the corresponding service compositions, which are often represented as business processes or as workflows of services, and which are typically deployed upon startup of the system. To maintain the target *service level agreement* (SLA) and continuous availability of services, multiple replicas of resources need to be deployed. This includes both back-end service nodes that execute the requests, as well as front-end nodes which are dedicated engines that invoke the corresponding services.

1.1.2 Cloud Computing Models for Service Providers

Cloud computing is an emerging computing paradigm, featuring elastic capacity provisioning and ease of operational management for a wide range of services. Resources, such as processors, storage, and network, are provided in an on-demand fashion to multiple service providers (i.e., clients of the cloud), who may deploy multiple services exhibiting disparate workload patterns. Essentially, cloud platforms enable resource sharing among multiple service providers, as well as among multiple services deployed by the same provider. Typically, the basic computing unit in compute clouds is the virtual node, on which different services can be deployed in an on-demand fashion. Depending on the specific cloud architecture, virtual nodes can correspond to either virtual machines or physical machines.

The NIST model of cloud computing [Mell and Grance, 2011] encompasses three different service models — Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Service providers can use four types of cloud deployment models — private cloud, community cloud, public cloud, and hybrid cloud. The first three deployment models are ordered from the least to the most dynamic in terms of provisioning elasticity, and the hybrid cloud is a mix of resources from two or more of the other models.

1.2 Deployment of Multi-Tier Services in Clouds

Various service replication strategies [Salas et al., 2006; Zheng and Lyu, 2008, 2009; Dustdar and Juszczyk, 2007] have been developed for fault-tolerant service-oriented systems. Often, only the replication of atomic services has been considered and the optimal number of replicas for composed services has been overlooked. Consequently, to optimize SLA and operational cost, the optimal provisioning of service replicas has mainly been shown in the context of simple *single-tier* web hosting systems [Lin et al., 2013] , i.e., clients send requests directly to services. For *multi-tier* web hosting systems, most existing studies [Singh et al., 2010; Zhang et al., 2008] design replication polices independently for each tier. In reality, the provisioning of front-end replicas depends on the performance of the second-tier service layer, due to the blocking I/O which is a result of the processing of consecutive service invocations within a composition. The performance of the service replicas depends on the invocations dispatched by front-end replicas and the corresponding load balancing among back-end service replicas. It is very challenging to dynamically provide resources in systems with multiple tiers, i.e., front-end and back-end service tiers, which encounter time-varying and -correlated workloads, such that the cost is minimized without compromising performance.

1.3 Deployment of Services in Public Cloud

Public clouds are clouds from which customers can obtain resources, e.g., virtual machines in compute clouds for provisioning web services. Deploying services in public compute cloud environments is an attractive solution, due to cost and ease of management advantages. In a public cloud, a set of preconfigured VM instances is available at different costs for different sizes, and their corresponding hardware-related performance metrics are provided at best effort [Ristenpart et al., 2009].

Hosting services in a cloud relieves the service provider from maintaining an expensive computing infrastructure. Thanks to on-demand virtual resource provisioning, cloud operators provide on-demand computing capacity, enabling elastic service provisioning. Another advantage of cloud environments is their pay-as-you-go billing feature. The service provider can thus request the necessary computing capacity in the unit of VMs from the cloud operator, according to the workload. Consequently, hosting services in a cloud — in conjunction with an effective service replication policy — can achieve significant cost savings for the service provider.

1.3.1 Performance Variability

When migrating various applications onto cloud platforms, one of the common weaknesses observed in public cloud environments is the higher performance variability compared to private platforms. In particular, VMs with the same specifications (i.e., incurring the same costs for the user) show significant performance variability in terms of throughput; some VMs are faster and some are slower. The observed higher performance variability also holds true for the response time, which fluctuates significantly, and tail latency degrades due to the heterogeneity of the underlying hardware and the workloads co-located on the same physical hosts. The effects of resource sharing that result from consolidating multiple VMs on the same physical hosts, are dynamically changing depending on varying workloads and on workload management actions taken by the cloud operator, such as VM consolidation and VM migration. Furthermore, hardware features such as dynamic frequency scaling can have an impact on performance depending on the workloads and on VM consolidations. As a consequence, the computing capacity of individual VMs fluctuates, and so does the aggregated capacity of all provisioned VMs of a service provider.

Although virtualization enables the efficient multiplexing of workloads across the ample hardware resource, performance isolation is limited, especially for applications that are not CPU intensive. While the performance variability persists in cloud platforms, little is known about the sensitivity of services on different VM configurations in terms of capacity, i.e., the maximum number of service requests that can be processed sustainably, and the aggregate impact of the capacity variability of a single VM on the QoS of the entire service cluster. VM provisioning of service systems is typically based on the average capacity, which in turn is a good indicator for systems experiencing low variability and providing simple *Quality of Service* (QoS) guarantees, such as average throughput over a certain threshold. To avoid performance penalties due to variability in the cloud, se-

lecting VMs with desirable performance becomes of paramount importance not only to reduce performance variability, but also to optimize cost. Consequently, empirical approaches are proposed to acquire VMs with higher capacities. However, due to the empirical nature of the proposed VM selection strategies, a QoS promise of satisfying a given target throughput is only attained at best effort. Moreover, the resulting cost minimization may be arbitrary, depending on the workload dynamics of the underlying cloud platform.

1.3.2 Cost Structure

In addition to the performance variability, another distinguishing difference between private systems and public cloud platforms is the cost structure and restrictions imposed by the billing contract. On a private platform, turning a VM on and off is not restricted by any billing contract, whereas VMs requested in a cloud are typically charged for pre-defined billing periods such as an hour. Therefore, in a cloud it can be wasteful to turn VMs on and off without considering billing constraints. Moreover, frequently turning VMs on and off may cause not only additional costs but also some capacity loss because of the time overhead associated with the VM control actions. System performance (i.e., service response times) can fluctuate greatly during the transition of turning VMs on and off.

On the one hand, cloud platforms provide several cost advantages for elastic service provisioning. On the other hand, system dynamics become much more complex than in private platforms and pose several new challenges. Purely workload-driven service replication policies have been shown effective on private platforms [Chen et al., 2005; Lin et al., 2013; Singh et al., 2010; Stewart et al., 2007], implementing simple control actions such as turning service replicas on and off. However, such policies can fall short in optimizing the trade-off between cost and performance in a cloud, due to the lack of consideration of the variability in VMs' performance and billing contracts. For example, in a cloud, a lower number of faster VMs may have the same aggregate capacity as a higher number of slower VMs, but typically cost less, particularly if the faster and the slower VMs are not distinguished by the billing contracts. To optimize service provisioning costs and service performance simultaneously, the service replication policy in the cloud needs to choose not only the right number of VMs but also the VMs with better performance. As such, a broad range of criteria, such as workload, heterogeneity of VM performance, and billing contracts, needs to be taken into consideration when designing service replication algorithms for cloud environments.

1.3.3 Tail Response Times

Several empirical studies [Xu et al., 2013; Schad et al., 2010; Casale and Tribastone, 2013] point out a common pitfall in clouds: the execution speed of an application within a virtual machine (VM) fluctuates significantly due to the heterogeneity of the underlying hardware and the workloads co-located on the same physical host. Although virtualization enables efficient multiplexing of workloads across ample hardware resources, performance isolation is limited [Chen et al., 2012; Björkqvist et al., 2013]. The resulting exogenous variability not only hampers the satisfaction of the users, but also results in non-negligible business losses associated with the violation of service level agreements (SLAs) often specified in terms of tail response times.

The degradation of tail response times in the cloud is further exacerbated when deploying cluster-based applications [Xu et al., 2013], i.e., relying on a large number of VMs. Web [Dean and Barroso, 2013] and big data services [Reiss et al., 2012] are typical examples requiring such cluster deployments. In addition to the modulated execution speed and cluster size, the distribution of response times, particularly the tail, is also affected by the load balancing algorithm distributing the load across VMs and the processor scheduling mechanism at each VM. Typically, a simple round robin algorithm is widely adopted, such as the one used in the Amazon EC2 cloud [EC2, 2014]. Requests are executed in a Processor Sharing (PS) fashion on individual VMs, which are typically hosted on separate physical servers. Overall, when deploying application clusters on today's cloud, three aspects are crucial for capturing the distribution of workloads and response times: the modulated execution speed of VMs, the load balancing algorithm, and the processor scheduling.

1.4 Load Balancing of Replicated Services

Since service systems are normally provisioned using multiple replicas, distributing the incoming load among the available resources is critical. By keeping the utilization level of the provisioned resources high, fewer resources are needed, leading to lower costs. A number of approaches for balancing the loads exist, and some are more suitable for certain scenarios than others.

Load balancing schemes can be categorized into two types: load oblivious and load aware. The former, such as random selection and round-robin selection, distributes requests to available front-end and back-end service replicas, independent of their loads. On the contrary, the latter dispatches requests de-

pending on the monitored loads of the front-end and back-end service replicas. Join the shortest queue (JSQ), where incoming requests are dispatched to the server with the least number of outstanding requests [Whitt, 1986], is one of such load balancing schemes, which are shown effective in systems with low variation in loads; however, its scalability is limited due to the implementation overhead of continuously monitoring the load. On the other hand, a dynamic lottery balancing scheme, combining the advantages of load oblivious and aware, is extensively applied for scheduling in operating system [Waldspurger and Weihl, 1994] and service system contexts [Mosincat and Binder, 2009].

To ensure scalability, today's web services are replicated and hosted on distributed systems that experience regular resource upgrades and are thus comprised of heterogeneous servers. The employment of virtualization technology further amplifies the server heterogeneity, especially on hosting platforms shared with different service providers such as computing clouds. Web service applications are characterized not only by disparate resource requirements (e.g., CPU-intensive browsing service vs. I/O-intensive transaction service), but also by time-varying request workloads [Zhang et al., 2005]. Consequently, the overall system workloads fluctuate in terms of mixes of applications and the volume of requests [Singh et al., 2010]. The heterogeneity of servers, together with diversified applications with different workload characteristics, further exacerbates the challenges of load balancing.

There is a large body of load balancing studies [Zhang et al., 2005; Cardellini et al., 1999; Cherkasova and Ponnekanti, 2000] that mainly focus on homogeneous systems and consider a single bottleneck resource where queues build up. The JSQ policy has been shown very robust theoretically [Gupta et al., 2007] and practically [Apache, 2014], in distributing the entire load across distributed servers. In a heterogeneous system experiencing time-varying application mixes, such a policy can potentially lead to the situation where servers receive similar amounts of requests but servers with powerful CPU (resp. disk) process a lot of IO- (resp. CPU-) intensive requests. Clearly, depending on the received application mix, the use of servers with different bottleneck resources can result in very different performance, such as response times. Therefore, it is imperative for the load-balancing policy to distribute the server load evenly as well as the resource load, which is influenced by the application mix received at individual servers.

1.5 Problem Statement

In this thesis, we consider service systems provisioned in the cloud, from a service provider's point of view. These systems provide services which, either by themselves or together with other services, form applications. Clients send requests to the applications, with strong time variability patterns. The application requests are directed to the appropriate entities, and on the way possibly broken down into multiple internal service requests, resulting in complex workloads. A high-level overview of the considered system can be seen in Figure 1.1.

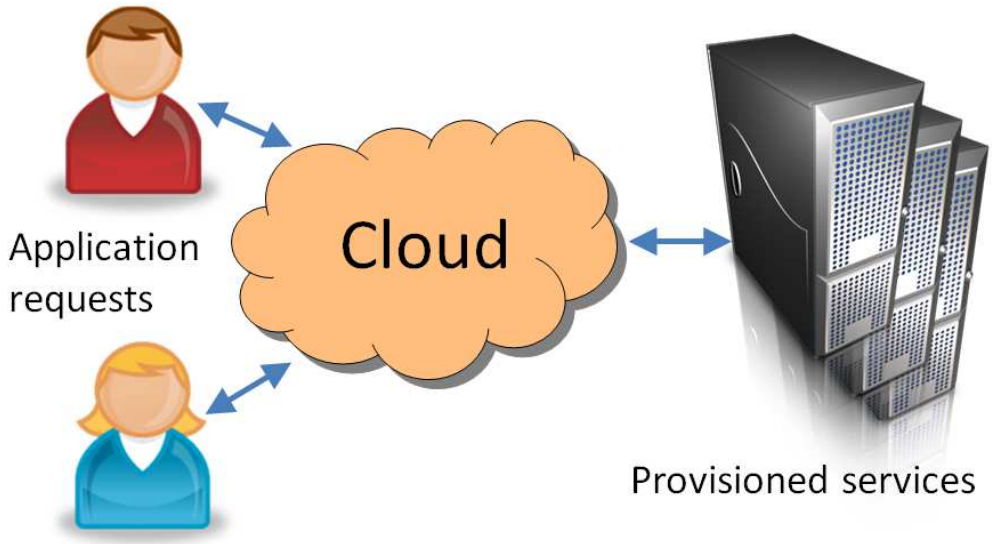


Figure 1.1. Application clients accessing services provisioned in the cloud

Essentially, an application is composed of services which are hosted on cloud VMs. Due to the nature of resource sharing and the trend of using heterogeneous hardware in the cloud, VMs tend to exhibit performance variability, i.e., VMs with the same specification experience different execution speeds. The main performance metrics of interest are resource utilization as well as the percentiles of throughput (in terms of requests per second) and response times, which are challenging to predict. All in all, the difficulty of resource management arises from the complexity of workloads, cloud systems, and the sophisticated performance metrics. The general research question that we try to answer is the following:

> ***How should service providers dynamically provision cloud resources, in terms of VMs, so that various service objectives, especially the high percentile throughput and response times, can be fulfilled in a cost effective manner?***

Furthermore, we examine different aspects of complexity in the aforementioned question — service provisioning in clouds — and dive into a subset of the problem space. In the following, we introduce the particular subproblems and argue for their relevance:

- **Two-tier Application Provisioning**

  Many service providers build their services using two or more tiers, e.g., front-end web servers and back-end databases. Understanding the interactions between the different tiers is critical for efficient provisioning. Tools and services exist for determining resource usage of individual replicas and for dynamically adjusting the amount of provisioned resources, such as the number of replicas, but for more complex, multi-tier deployments they might lead to underutilization of resources, underprovisioning of unidentified bottlenecks, or both. The major challenge with provisioning of multi-tiered systems is understanding the interaction and dependencies between the different tiers, and their impact on provisioning decisions. For this particular problem, the research question we want to answer is:

  How should resources for two-tier cloud applications be efficiently provisioned?

- **Opportunistic Provisioning**

  The cloud infrastructure providers host the VMs on a plethora of different types of physical servers, dispersed among multiple data centers in different parts of the world. This often means that the physical hardware running two identical VMs can vary greatly, and this can manifest as performance variability for the cloud service providers. Provisioning for different application and service types requires understanding of the resource usage, and its variance over time. When independent application and service types are operated by the same service provider, there is potential for exploiting the different usage patterns in terms of more efficient resource usage and cost savings. One big challenge for cloud service providers is therefore how to efficiently provision multiple applications and services when faced with performance variability of VMs. To address this issue, the research question becomes:

  How can cloud service providers take advantage of performance variability in the cloud when provisioning multiple applications and services?

- **Tail Throughput in the Cloud**

  Cloud services are often governed through Service Level Agreements that defined the Quality of Service that is to be provided. An example of such a requirement is that the throughput shall be above a certain number of requests per second for a certain percentage, e.g., 99% of the time. Provisioning for average throughput requirements is relatively straight-forward, but provisioning resources in a way that satisfies tail throughput requirements in a cost-efficient manner is not a trivial task. Therefore, our research question related to the tail throughput is:

  How should cloud service providers provision resources to efficiently provide tail throughput guarantees?

- **Tail Response Times in the Cloud**

  For interactive cloud services, an even more important performance metric than the throughput is the response time. As is the case for the throughput, predicting and provisioning for the average response time is not overly difficult. However, predicting the tail response time is a very complex and difficult problem:

  How should cloud service providers provision resources to efficiently provide tail response time guarantees?

1.6 Outline of Proposed Solution

Due to the complexity of the system, e.g., multiple application and service types, multiple replicas, performance variability, and workload variability, it is not possible to address all possible aspects in a single, comprehensive setting. Thus, we resort to considering different subproblems in isolation. This is also the approach taken in other studies, but the related work often falls short by considering oversimplified scenarios, e.g., in terms of architecture, system model, or performance requirements. Our focus is on models and algorithms that are evaluated on simplified scenarios, but which can be expanded to accommodate more details obtained from real systems.

A summary of how our approach of tackling subproblems in isolation has been structured into separate chapters is shown in Table 1.1. We first focus on tackling complex workloads, i.e., time-varying composed services in two-tier systems. Thereafter, we consider the challenges introduced by the system complexity that

Chapter	Workload	System	Performance
3	Composed services	Two-tier systems, homogeneous servers	Simpler performance metrics; utilization
4	Atomic services, multiple service types	Spatial variability, i.e., heterogeneous servers	Best effort
5	Atomic services	Temporal variability	Tail throughput
6	Atomic services	Temporal variability	Tail response time

Table 1.1. Summary of chapters.

is inherent, but not limited to, the cloud, i.e., spatial and temporal performance variability of servers. Finally, we focus on deriving sophisticated performance metrics, particularly for the tails, i.e., tail throughput and distribution of tail response times.

To handle workloads and application requests that vary over time, the service provider must be able to adjust the amount of resources used to provide the services. Figure 1.2 shows an example of a service system with two tiers, serving application requests for composed services. We investigate how to efficiently provision such two-tier service systems in clouds in Chapter 3. An example of a two-tiered service system is a front-end web server that serves requests by querying a back-end database server. We analyze the system from the point of view of a service provider that offers multiple different types of composed services. As service demands fluctuate, the service provider needs to adjust the amount of resources used to provision the services, while taking into account the dependencies between the system tiers. To simplify the analysis of the two-tiered service system scenario, we assume homogeneous servers that can be turned on or off whenever necessary.

While the initial work is agnostic of the underlying platform and assumes that all replicas run on identical systems, the work described in Chapter 4 looks at how to optimize service provisioning in public clouds. To efficiently provide cloud computing services in public clouds, the cloud service providers co-locate multiple tenants on the heterogeneous hardware located in data centers around the globe. Figure 1.3 shows a scenario where multiple identical servers, in terms of specification and cost, are used to provision services on top of public cloud computing platforms. In practice e.g., the underlying hardware and co-located workloads, mean that the observed performance is not always identical. For a service provider providing services, this presents both challenges and opportu-

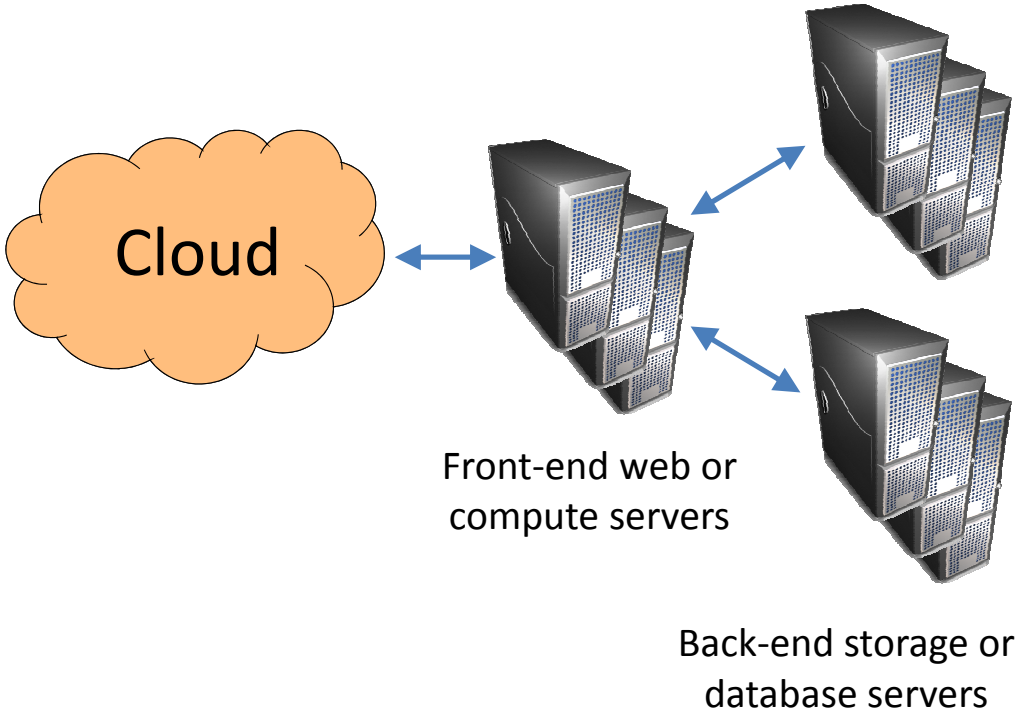


Figure 1.2. Service provisioning in multiple tiers, e.g., front-end and back-end

nities. We investigate how a service provider can leverage the observed performance variability in order to achieve better performance, in terms of throughput capacity, at a lower cost. We simplify the problem by looking at systems with single-tiered services of multiple different service types. Another assumption that we make is that while the performance may be different for individual VMs provisioned on the public cloud platform, the performance of a VM does not vary over the observation period. The shorter the observation period, the more valid the assumption.

The analysis in Chapter 4 assumes that the performance variability among VMs with identical specifications is constant over time, and shows how to effectively take advantage of this. However, the observed performance variability in public cloud platforms may also vary over time, e.g., due to VM migrations, or varying system utilization of VMs co-located on the same physical machine. In Chapter 5 we first show the performance impact that co-located VMs can have on a VM running a wiki service, and thereafter attempt to take this temporal performance variability into account when deciding on how to provision services in the cloud. We model the system using a Markov-chain model, and further show that provisioning a system based on the observed average capacity fails to

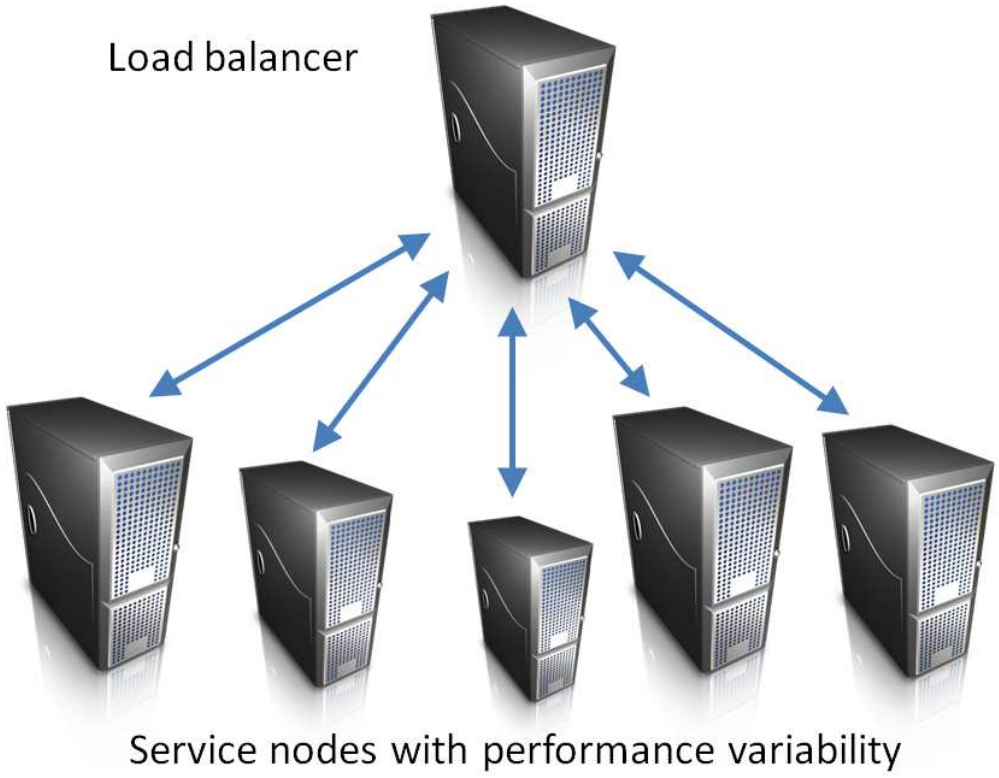


Figure 1.3. Service nodes with performance variability

avoid tail performance degradation, which has an impact on the fulfillment of QoS promises.

While it is useful to be able to provision systems with QoS guarantees based on the throughput capacity, it is more common for interactive web services to have QoS SLAs written in terms of response times. Providing tail response time guarantees (e.g., the response time for 99.99% of requests must be below $2s$), however, is a more complex problem to solve. In Chapter 6, we approach the problem using large deviation analysis and use an approximation scheme to obtain the tail response times, which are then used for provisioning decisions.

1.7 Contributions

This dissertation is based on several published and submitted pieces of work. The contributions regarding the provisioning of two-tier services in the cloud (Chapter 3) are threefold: (1) a model and analysis capturing many key features of two-tier service-oriented systems: time-varying workloads, execution paral-

lelism, and the inter-dependency between the two tiers; (2) a novel replication policy, simultaneously controlling the number of provisioned resources in the two interdependent tiers, based on monitored workload and performance metrics; and (3) bounding analysis on effective and nominal utilization for resources in the first tier. This work was published in:

- Mathias Björkqvist, Lydia Y. Chen, and Walter Binder. Dynamic replication in service-oriented systems, *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, IEEE Computer Society, pp. 531-538.

The original scientific contribution of the work on leveraging performance variability for service provisioning in public clouds (Chapter 4) is a novel service replication policy, which is specially designed to explore the temporal variability of VM performance on public cloud platforms. Compared to replication policies oblivious to the unique characteristics in public clouds, e.g., performance variability, pay-as-you-go billing periods, the proposed opportunistic replication policy is shown to achieve lower cost and better performance when optimizing not only for a single service type, but for a service providers entire set of resources provisioned in a public cloud. This work was published in:

- Mathias Björkqvist, Lydia Y. Chen, and Walter Binder. Opportunistic service provisioning in the cloud. *Proceedings of the 2012 5th IEEE International Conference on Cloud Computing (CLOUD)*, pp. 237-244.

The contributions related to providing throughput QoS guarantees for services provisioned in public clouds (Chapter 5) are twofold: (1) quantitative characterization of the capacity variability of a VM running a wiki service when colocated with another VM running various different workloads, and (2) a Markov-chain model to avoiding tail throughput performance degradation. Based on our experiments and model, a cluster of VMs can be properly dimensioned using appropriate VM configurations, such that the best trade-off between cost and throughput QoS fulfillment is achieved. This work was published in:

- Mathias Björkqvist, Sebastiano Spicuglia, Lydia Y. Chen, and Walter Binder. QoS-Aware Service VM Provisioning in Clouds: Experiences, Models, and Cost Analysis. *Service-Oriented Computing*, Springer, pp. 69-83.

In the work on optimizing for tail response times for services provisioned in public clouds (Chapter 6), the contributions can be summarized as follows: First,

the workload distribution is derived for hard-to-analyze systems that capture the key characteristics of today's cloud systems, i.e., renewal arrivals, highly varying job sizes, Markov-modulated execution speeds, processor sharing, and round-robin load balancing. Second, an approximation scheme for the tail response times is developed, as these are one of the critical SLA parameters, and this is used to further optimize the cluster size.

During the course of the PhD studies, I was also involved in related work resulting in the following publications:

- Mathias Björkqvist, Lydia Y. Chen, Marko Vukolić, and Xi Zhang. Minimizing Retrieval Latency for Content Cloud. *Proceedings of 2011 IEEE INFOCOM*. pp. 1080-1088.
- Mathias Björkqvist, Lydia Y. Chen, and Walter Binder. Optimizing service replication in clouds. *Proceedings of 2011 Winter Simulation Conference*. pp. 3312-3322.
- Mathias Björkqvist, Lydia Y. Chen, and Walter Binder. Cost-driven Service Provisioning in Hybrid Clouds. *Proceedings of 2012 IEEE Service-Oriented Computing and Applications (SOCA)*. pp. 1-8.
- Sebastiano Spicuglia, Mathias Björkqvist, Lydia Y. Chen, Giuseppe Serazzi, Walter Binder, and Evgenia Smirni. On load balancing: a mix-aware algorithm for heterogeneous systems. *Proceedings of 2013 ACM/SPEC International Conference on Performance Engineering (ICPE)*. pp. 71-76.
- Robert Birke, Mathias Björkqvist, Lydia Y. Chen, Evgenia Smirni, and Ton Engbersen. (Big)data in a virtualized world: volume, velocity, and variety in cloud datacenters. *Proceedings of 2014 USENIX Conference on File and Storage Technologies (FAST)*. pp. 177-189.

Chapter 2

State of the Art

Related studies exist in all the different areas related to service provisioning in clouds. Section 2.1 explores work done regarding replication of service systems in clouds, data centers, or similar scenarios, and both atomic and composed services are addressed. Studies on service provisioning in public clouds in Section 2.2 deal with QoS, cost optimization, and the performance variability observed specifically in public clouds. Section 2.3 summarizes relates work in the field of modeling and optimizing for tail response times in clouds. Related studies in Section 2.4 cover most related aspects of load balancing of service systems.

2.1 Replication of Services in Clouds

The related work regarding replicated web services in the cloud context is mainly discussed in two contexts: fault tolerant services, and resource provisioning.

Fault-tolerant services: In order to provide highly dependable service-oriented systems, various service replication framework and strategies have been developed in different system scenarios. Many studies [Salas et al., 2006; Zheng and Lyu, 2008, 2009; Dustdar and Juszczyk, 2007] consider the replication of atomic services and do not address the issues of optimal number of replicas. Salas et al. [Salas et al., 2006] developed a replication framework, WS-Replication, which enables the deployment in a set of sites. In particular, WS-Replication respects web service autonomy and exclusively uses SOAP to interact across sites via WS-Multicast. Zheng and Lyu [Zheng and Lyu, 2008, 2009] compare different combinations of passive and active replication strategies, using their proposed evaluation framework. Their focus is on selecting a suitable replication strategy such that the performance threshold and failure threshold are met. Dustdar and Juszczyk [Dustdar and Juszczyk, 2007] studied service replication strategies on

mobile ad-hoc networks, whose topologies vary over time. They developed a passive replication strategy and validated it on a simulation prototype.

In contrast, You et al. [You et al., 2009] consider the replication for composed services. They replicate services that have the longest response time and deploy them on the nodes with maximum available capacity. Via simulation, their proposed strategy decreases the response time of composite service as well as balances the load. We consider both the replication of the composition execution engine and the atomic services, and focus on deriving the optimal number of replicas for optimizing system resources and performances.

Resource efficient services: To design a scalable and cost-effective service-oriented system, dynamic resource provisioning is very critical, especially when encountering time-varying workloads. A comparison of different web service provisioning architectures is presented in [Pautasso et al., 2008]. A number of studies [Chen et al., 2005; Lin et al., 2013] focus on a single tier web server system, whereas others [Singh et al., 2010; Zhang et al., 2008; Stewart et al., 2007] address multi-tier web server systems. Resource provisioning strategies in multi-tier systems often consider each tier independently from other tiers. Petrucci et al. [Petrucci et al., 2011] implement a dynamic service provisioning policy to optimize power consumption on a heterogeneous cluster. While most provisioning studies monitor the request rate, Singh et al. [Singh et al., 2010] monitors not only the request rate but also the mix of applications. Pautasso et al. [Pautasso et al., 2007] design an engine for autonomic resource provisioning that can dynamically reconfigure resources to different tasks as conditions change. The autonomic, self-optimizing replica placement module proposed by Serrano et al. [Serrano et al., 2008] dynamically places data copies on servers close to clients.

2.2 Provisioning of Services in Public Clouds

Cloud computing is an emerging platform for commercial and scientific applications, due to advantages in the pay-as-you-go business model and elasticity capacity provision. The services offered by public clouds are a good fit for applications and services with growing or fluctuating demand, as the provisioned capacity can be adjusted based on workload observations or predictions.

To better understand how to best go about provisioning services and applications in the cloud, many studies have looked at the replication aspect — how are replicated services best provisioned in the cloud, how are they migrated from existing systems, and what are the differences compared to local data centers or clouds. The observed performance variability has been investigated from many

points of view, such as architectural and platform differences, as well as the impact of co-located VMs. For deciding whether or not to move a system into the cloud, the actual migration from existing systems also needs to be taken into account. Guaranteeing reliability and performance for services provisioned in the cloud is also not always straight-forward, and related studies also look at the QoS-aspects of such services.

Migrating services to the cloud: The focus of related studies regarding migration of service-oriented applications from existing systems to the cloud is widely spread: From a summary of the practical experiences [Chauhan and Babar, 2011], to frameworks for automating and easing the migration process [Mietzner et al., 2009], and cost optimization [Trummer et al., 2010; Fehling et al., 2010]. Chauhan and Babar [Chauhan and Babar, 2011] report practical experiences of migrating an open source software framework, Hackystat, to the cloud. One of the key findings is that it is easier to migrate software systems consisting of stateless components to IaaS clouds.

Performance variability: Various studies [Kossmann et al., 2010; Jackson, Ramakrishnan, Runge and Thomas, 2010; Jackson, Ramakrishnan, Muriki, Canon, Cholia, Shalf, Wasserman and Wright, 2010; Ueda and Nakatani, 2010; Nurmi et al., 2009] present performance studies and report their experiences on migration of various applications to commercial cloud platforms. A common observation is the high variability in the quality of service. Kossmann et al. [Kossmann et al., 2010] present a comprehensive evaluation of database applications under different cloud architectures. They conclude that the cost and performance of the services vary significantly depending on the workload. Jackson et al. [Jackson, Ramakrishnan, Muriki, Canon, Cholia, Shalf, Wasserman and Wright, 2010; Jackson, Ramakrishnan, Runge and Thomas, 2010] port various scientific applications, such as SNFactory pipeline, to Amazon [EC2, 2014]. Their results show that the performance of EC2 is more variable and slower than non-cloud computing platforms, due to the limitation of interconnects on EC2. Ueda and Nakatani evaluate a wiki workload and Apache daytrader using two open-source cloud platforms, OpenNebula [OpenNebula, 2015] and Eucalyptus [Nurmi et al., 2009]. The two platforms give very different performance results, e.g., in terms of VM provisioning, response time, and throughput, compared to Amazon EC2.

Most existing studies on the performance variability of applications hosted in the cloud are based on empirical experiments, especially in terms of average and 95^{th} percentile response time [Xu et al., 2013; Schad et al., 2010], and aim to discover the root cause of such a phenomenon [Kossmann et al., 2010; Jackson, Ramakrishnan, Runge and Thomas, 2010; Mao and Humphrey, 2012]. The observations made from cloud experiments are mainly based on a single type of

configuration and simple benchmarks. A few studies [Spicuglia et al., 2013; Xu et al., 2013; Schad et al., 2010] focus on multiple types of VM configurations and try to quantify the variability in their response times. Moreover, the variability in throughput is largely evaluated under a particular workload intensity, instead of using the maximum sustainable throughput, i.e., the capacity.

Reliability: Various service replication strategies have been developed and evaluated for guaranteeing the reliability [Zheng and Lyu, 2009; Dustdar and Juszczyk, 2007] or performance under time-varying workloads [Chen et al., 2005; Lin et al., 2013; Singh et al., 2010; Stewart et al., 2007; Petrucci et al., 2011]. To deliver highly dependable service systems, Zheng and Lyu [Zheng and Lyu, 2009] compare different combinations of replication strategies, using their proposed evaluation framework. Their objective is to select a suitable strategy such that the performance threshold and failure threshold are met. Dustdar and Juszczyk [Dustdar and Juszczyk, 2007] developed a passive replication strategy on mobile ad-hoc networks, whose topologies vary over time, and validated it on a simulation prototype. As for workload driven replication strategy, both single-tier [Chen et al., 2005; Lin et al., 2013] and multiple-tier [Singh et al., 2010; Stewart et al., 2007] web server systems in a non-cloud platform are well addressed. Petrucci et al. [Petrucci et al., 2011] implement a dynamic service provisioning policy to optimize power consumption on a heterogeneous cluster. While most provisioning studies monitor the request rate, Singh et al. [Singh et al., 2010] monitor not only the request rate but also the mix of applications.

Consistency: Data consistency is another aspect that becomes more relevant in public clouds. There is often a non-trivial trade-off between cost, consistency, and availability [Kraska et al., 2009]. Weak consistency is often considered sufficient for applications deployed in public clouds [Fetai and Schuldt, 2012]. This weak consistency may result in increased operational costs, e.g., due to overselling products in a web shop. On the other hand, providing stronger consistency comes at the expense of higher costs, both in performance and monetary terms. These aspects also need to be taken into account when deciding whether or not to deploy applications in the cloud.

QoS: Recent studies on QoS analysis for cloud services [Zheng et al., 2011; Ye et al., 2012; Tsakalozos et al., 2011] are mainly driven by service compositions and service selection, using a Markovian decision process [Ramacher and Mönch, 2012] or a Baysian network model [Ye et al., 2012]. In contrast, studies focusing on constant QoS value, e.g., Zheng et al. [Zheng et al., 2011] proposed a calculation method to estimate the probabilistic distribution of QoS. Toffetti et al. [Toffetti et al., 2010] use Kriging surrogate models for approximating the performance profile of virtualized, multi-tier Web applications, including analyz-

ing collected data to diagnose potential SLA violations. However, the impact on the QoS due to the underlying performance variability of the cloud is to a large extent overlooked.

Reducing variability: Meanwhile, another set of studies focus on developing solutions to reduce the performance variability in a best effort manner, from the perspective of service providers. Particularly, Farley et al. [Farley et al., 2012] propose opportunistically selecting VMs which have high capacity, while discarding VMs with low capacity. Another type of solution is to try to figure out the underlying hardware and neighboring workloads, so as to select similar physical hosts [Schad et al., 2010] and influence the neighboring VMs [Ristenpart et al., 2009]. As the methodology is trial and error, the QoS of the target application, e.g., the service availability, is not always guaranteed. Moreover, the cost analysis is over-simplified, without considering the performance variability. CopperEgg provides a tool [AWS Sizing Tool, 2014] that tracks your current system usage and recommends optimal Amazon EC2 instance sizes. Whether or not they take the observed variability of VMs into account is not clear, as it is not shown in any of the metrics in the product presentation.

2.3 Tail Response Times

In this section, we discuss the related work in two directions: the modeling work on predicting the tail response times for complex queueing systems that show a great resemblance to real systems, and the optimization work that tries to mitigate the response time degradation due to exogenous variability in the cloud.

2.3.1 Modeling Tail Response Times

Given a vast amount of research on estimating the average response times, obtaining the entire distribution of response times is no mean feat for non-Markovian systems, in terms of arrival and service processes. An example of this are $M/G/1/PS$ queueing systems [Kleinrock, 1975; Gautam, 2012], which are widely adopted to model various computing systems executing highly varying job sizes with a fixed speed in a processor sharing discipline. The introduction of cloud computing pinpoints another dimensionality of the modeling challenges, i.e., varying execution speed. The state-of-the-art deals with two causes of varying execution speeds: state-dependent and exogenous/environmental variability. The former [Gupta and Harchol-Balter, 2009; Rege and Sengupta, 1985; Zhang and Zwart, 2008] models the execution speed based on the current state of the system, i.e., the

combination of the number of jobs and the multiprogramming levels. The latter [Mahabhashyam and Gautam, 2005; Casale and Tribastone, 2013; Zhang and Zwart, 2012; Dorsman et al., 2013] models the transition of execution speeds as Markov-modulated speed for single queue and multiple queues.

In terms of the impact of arrival process, prior studies efficiently model the average response times of bursty workloads using a Markovian Arrival process [Casale et al., 2008; Sansottera et al., 2013], particularly for multi-tier applications [Mi et al., 2008]. The Markov-modulated execution speed for single queues is discussed in [Gautam, 2012; Baykal-Gursoy and Duan, 2006; Boxma and Kurkova, 2001]. While both Zhou and Gans [Zhou and Gans, 1999], as well as Boxma and Kurkova [Boxma and Kurkova, 2001], study two execution speeds, they employ different assumptions on when the changes of execution speed take place, i.e., only after the completion of job execution and during the job execution. Moreover, Boxma et al. consider an $M/G/1$ queue where the speed of the server alternates between two values with high speed periods having exponential distribution and low speed periods having a general distribution. Another related article is one by Massey [Massey, 2002], where the author focuses on deriving the queue process and the time-varying aspect, in particular an $M_t/M_t/1$ queueing system. In [Mahabhashyam and Gautam, 2005], the authors generalize the execution speed to any Markov process, and also the tail distribution of the workload in steady state. However, all the aforementioned analysis requires the arrival process to be Markovian.

There are a few studies that consider multiple queues with Markovian-modulated speeds. Dorsman et al. [Dorsman et al., 2013] obtain the heavy-traffic approximation for a steady distribution of workloads, queue lengths, and response times for parallel queueing networks with Markov-modulated service speeds, by combining a functional central limit theorem approach and matrix-analytic methods. However, the impact of different traffic intensities and renewal arrivals are not considered there. To efficiently approximate the mean performance indexes, i.e., throughput and response time, Casale et al. [Casale and Tribastone, 2013] propose a generalized blending algorithm to model any number of execution speeds experienced by servers in the cloud. They are based on solving the ordinary differential [Kurtz, 1970] that defines an approximate transient analysis method for queueing network models. Their approach is limited to queueing networks with Markovian arrivals and Coxian jobs size distribution.

To the best of our knowledge, the tail distribution of response times of many queue scenarios with execution speed modulated according to exogenous environmental processes and renewal arrivals are yet to be explored.

2.3.2 Optimizing for Tail Response Times in Cloud

Recognizing the importance and challenges of mitigating the performance variability in cloud computing, various reactive and proactive optimization strategies have been proposed. The reactive strategies center on obtaining a set of VMs that are of the same configurations but provide better performance, i.e., faster execution speed. To such an end, Farley et al. [Farley et al., 2012] use testbed experiments whereas Björkqvist et al. [Björkqvist et al., 2012] leverage simulation. Krebs et al. [Krebs et al., 2014] quantify the performance isolation of cloud-based systems using different metrics. Kraft et al. [Kraft et al., 2011] model the impact of workload consolidation on VM disk IO response times. Schad et al. [Schad et al., 2010] focus on the exogenous variability resulting from the underlying heterogeneous hardware and develop strategies to select VMs that are hosted on the same platform as to reduce the variability of the execution speeds. As for proactively mitigating the performance degradation, Björkqvist et al. [Björkqvist et al., 2013] leverage a continuous Markovian model to capture the distribution of tail throughput and further optimize the cloud cluster that fulfills the target tail throughput at minimal cost. Sansottera et al. [Sansottera et al., 2012] provide a consolidated model that considers power, performance and reliability aspects when estimating the impact of hardware virtualization on the operational cost and performance of data centers. The model developed by Casale et al. [Casale and Tribastone, 2013] is meant to enable efficient exploration of the decision space for cloud deployments, but with focus on the average performance index.

In summary, the optimization strategies for tail response times fall short in providing SLA guarantees, i.e., they only promise best effort. Our study adopts the proactive approach to model various important aspects of cloud clusters and further optimize the cluster size so that the tail response times specified in SLAs are statistically guaranteed.

2.4 Load Balancing of Service Clusters

The related work in the area of load balancing of service clusters can be divided into two areas: composed and atomic services. *Composed* services deal with scenarios where a single client invocation of a composed service leads to an execution of one or more executions of other services that provide functionality necessary for being able to complete the composed service. The order and number of times each individual service is executed depends on the composed service request. In studies on load balancing of composed services, the binding of ser-

vice replicas is one of the key issues investigated. For *atomic* services, the related work mainly focuses on how to distribute load between service replicas that are, for the most part, functionally equivalent. More specifically, addressed issues include heterogeneous hardware, separating incoming requests based on parameters such as the size, and to what degree the time-varying aspect of incoming requests is taken into account.

2.4.1 Load Balancing of Atomic Services

There is a large body of related studies of load balancing for various conventional service systems [Cardellini et al., 1999; Cherkasova and Ponnekanti, 2000; Riska et al., 2002; Zhang et al., 2005] and modern cloud systems [Dejun et al., 2011].

Cardellini et al. [Cardellini et al., 1999] qualitatively classified existing load balancing schemes at web server systems into four approaches, namely client-based, DNS-based, dispatcher-based, and server-based. Cherkasova and Ponnekanti [Cherkasova and Ponnekanti, 2000] developed FLEX, a locality-aware load balancing solution, especially for efficient memory usage. Riska et al. [Riska et al., 2002] proposed a load balancing scheme where incoming requests are sent to replicated server back-ends based on the request size. Zhang et al. [Zhang et al., 2005] expands on the earlier work [Riska et al., 2002]. Dejun et al. [Dejun et al., 2011] benchmark individual VMs obtained from cloud providers such as Amazon [EC2, 2014]. This information is then used to both balance the load within a tier (e.g., front-end or database), as well as to decide which tier a new VM is best suited for. Singh et al. [Singh et al., 2010] leveraged the idea of application mixes and proposed a mix-aware resource allocation for data centers. Most of the aforementioned studies focus on balancing loads on homogeneous servers with a single resource type, i.e., CPU.

2.4.2 Load Balancing of Composed Services

There is also a substantial amount of work done in the field of dynamic binding of composition execution engines and on scheduling algorithms. Most of the related work in the area addresses dynamic binding for compositions expressed in BPEL, that is, for business processes, since BPEL is a de facto standard and there are many implementations of BPEL engines. While BPEL supports dynamic binding by partner link assignment, it is neither possible to add new services at runtime, nor to change the service selection algorithm at runtime. Furthermore, in BPEL, dynamic binding is coupled with process business logic.

VieDAME [Moser et al., 2008] is a service monitoring and selection system based on aspect-oriented programming that intercepts SOAP messages and is able to dynamically replace services used in a business process. Dynamo [Baresi et al., 2007] relies on an aspect-oriented engine extension of ActiveBPEL engine to support monitoring and failure recovery. RobustBPEL2 [Ezenwoye and Sadjadi, 2008] provides dynamic binding with proxies. An approach to optimize system performance, taking hardware resources into account, is presented in [Zhang et al., 2007]. In [Mosincat and Binder, 2009], BPEL processes are automatically transformed to interact with a separate system that handles dynamic binding. Lottery scheduling in operating systems is presented in [Waldspurger and Weihl, 1994].

Chapter 3

Provisioning of Two-tier Services in the Cloud

Service-oriented systems, consisting of atomic back-end services utilized by front-end servers to provide composed services, are commonly deployed to deliver web applications. As the workloads of applications fluctuate over time, it is economical to autonomously and dynamically adjust system capacity, i.e., the number of replicas for back-end and front-end services. In this chapter, we propose a novel replica provisioning policy which adjusts the number of front-end and back-end service replicas periodically based on the predicted workloads, such that all replicas are well utilized at the target values. In particular, our proposed replica provisioning policy models the workload balance and dependency between front-end and back-end service replicas by estimating the probability that threads of front-end replicas are not blocked by I/O. Moreover, we derive the analytical bounds of effective front-end replica utilization and illustrate the cause of low nominal utilization at front-end replicas. We evaluate our proposed replica provisioning policy on a simulated service-oriented system, which hosts front-end and back-end service replicas on multi-threaded servers. The evaluated workload is derived from utilization traces collected from production systems. Through simulation, we demonstrate that our proposed replica provisioning policy effectively reduces the number of required replicas, while maintaining target utilization and lowering the response times of requests.

Our proposed replica provisioning policy dynamically adjusts the number of front-end and back-end service replicas periodically in *slotted control windows*. The workload and performance statistics are monitored in a control window of predefined length, and our replica provisioning policy aims to maintain the target utilization of the front-end and back-end service replicas by using the obtained

measurement data to adjust the number of service replicas. In particular, we estimate the nominal and effective utilization of front-end service replicas, which explicitly factors in inter-dependency among front-end and back-end service replicas using the derived non-blocking probability at front-end service replicas. Our simulation results show that our proposed replica provisioning achieves cost-effective provisioning of replicas, whose effective utilization is well maintained at the target value, and which deliver satisfactory end-to-end response time.

The contributions of this chapter are threefold; first, our model and analysis of two-tier service-oriented systems capture many key features: time-varying workloads, parallelism of replicas, i.e., thread pools, and the dependency between front-end and back-end service replicas. Second, we develop a novel replication manager, which considers the aforementioned features and dynamically controls front-end and back-end service replicas based on the monitored workload and performance metrics. Third, we provide bounding analysis on effective and nominal utilization for front-end service replicas, which essentially need to be kept less utilized than the back-end service replicas by a factor of the non-blocking probability of the front-end service replica threads.

This chapter is organized as follows: The system architecture is explained in Section 3.1. The service replication manager and service selection policy are described in Section 3.2 and Section 3.3, respectively. Section 3.4 contains the experimental results. The assumptions and limitations of this work are detailed in Section 3.5, and Section 3.6 summarizes the chapter.

3.1 System Model

In this chapter we consider a service-oriented system as depicted in Figure 3.1. *Composed services*, built using one or more atomic back-end services, are deployed and provided by the front-end servers and exposed through service interfaces to various client applications. When a service composition is invoked by a client, the front-end creates an instance of the composition and executes it. During the execution, atomic back-end services are invoked. We assume a service provider that hosts both the service compositions (in a front-end) and the atomic back-end services that are invoked when compositions are executed. We assume that client do not directly invoke the atomic back-end services; clients only invoke the exposed service compositions. In our model, both the front-end and back-end services can be replicated.

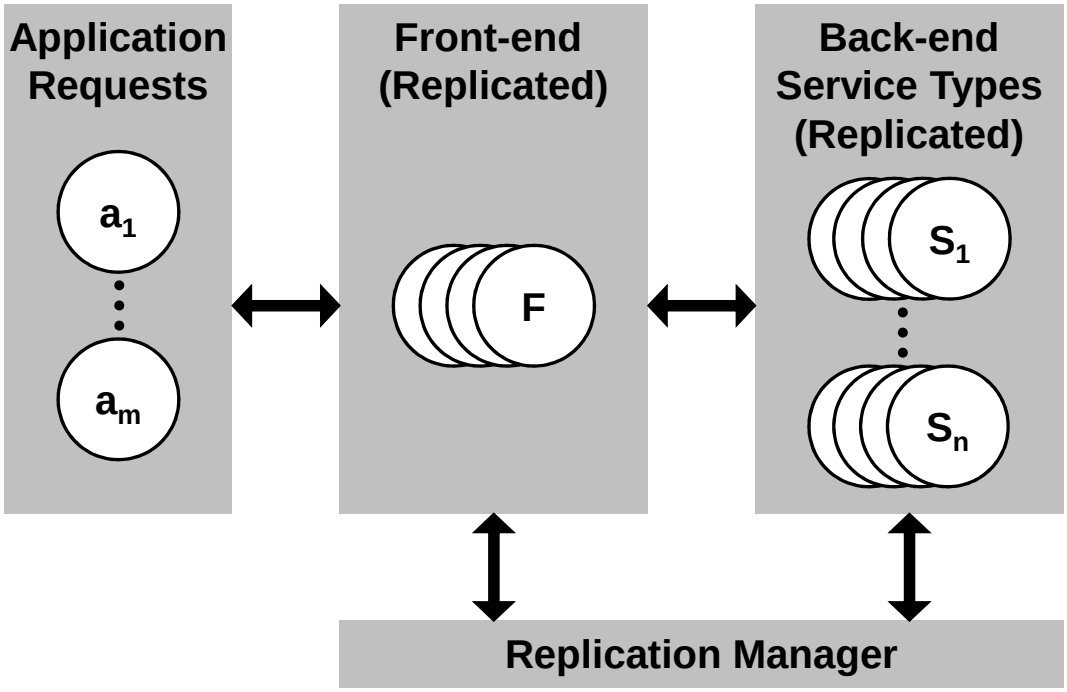


Figure 3.1. Architecture overview

3.1.1 Client Requests and Composite Services

For each service composition deployed in the front-end, we assume that the client requests generated from different applications may consist of disparate service compositions and have different workload characteristics (i.e., time-varying arrival rates). Here we consider only sequential service compositions where atomic back-end services are invoked in a given order. Here we do not model different workflow patterns [van Der Aalst et al., 2003] such as parallel split. For example, consider a system with two types of atomic back-end services, denoted by S_1 and S_2. Two possible service compositions are $\langle S_1, S_2 \rangle$ and $\langle S_2, S_2 \rangle$. For the first composition, S_1 is first invoked and then S_2, whereas in the second composition, S_2 is invoked twice consecutively. Each service composition corresponds to an application a, and Ω_a denotes the sequence of service invocations for application a. For the two examples above, $\Omega_a = \langle S_1, S_2 \rangle$, respectively $\Omega_a = \langle S_2, S_2 \rangle$.

3.1.2 Atomic Back-end Services

The system hosts I types of atomic back-end services, subscripted by $i \in \{1 \dots I\}$. There may be multiple replicas, n_{s_i}, for each service type. All service types are

considered stateless, i.e., for each invocation of an atomic back-end service by a front-end replica, a different replica may be bound. Each back-end replica has a queue for incoming requests (i.e., service invocations by the front-end) and maintains a thread-pool of fixed size, t_{s_i}, for processing these requests. Concurrent service invocations can be processed in parallel as long as there are threads available, i.e., sequential code sections in service replicas are not modeled.

We model each back-end service replica as a queueing system with one queue and multiple servers, each of which represents a single core/thread. An active replica processes service invocation requests sent by a front-end replica in a first come, first served (FCFS) manner, and the service i execution time per thread is a random variable with mean d_{s_i}. The response time of a service invocation is the sum of the queueing time and the execution time. We denote the average response time of service i by R_{s_i}.

3.1.3 Front-end Service Replicas

There are n_f front-end replicas, each of which is a distributed queueing system. A front-end replica queues incoming client requests that are then processed in a FCFS order. We let the average queueing time at a front-end replica be Q_f. The front-end replica has a thread-pool of fixed size, t_f, for executing client requests in a parallel fashion. Each front-end replica thread processes one request after the other, executing the corresponding service composition. For an invocation of service i, each thread selects a replica of service i, according to a load balancing scheme. The average execution time for a front-end replica to process a service invocation is assumed d_f. The invocations of atomic back-end services are handled using blocking I/O: a single thread is used for the entire execution of an instance of a service composition, and this thread will block while waiting for the results of invoked atomic back-end services. Since each instance of a service composition is executed sequentially by a single thread, we model only sequential compositions.

3.1.4 Replication Manager

The replication manager determines the number of front-end replicas, $n_f(t)$, and the number of back-end service replicas for each service type, $n_{s_i}(t)$, in discrete time windows of fixed length. We assume that each replica is deployed on a separate machine (i.e., resource contention between replicas on the same node need not be considered in this simplified model). In total, the provider has N machines to host all replicas. For all the windows, $n_f(t) + \sum_i n_{s_i}(t) \leq N$. The

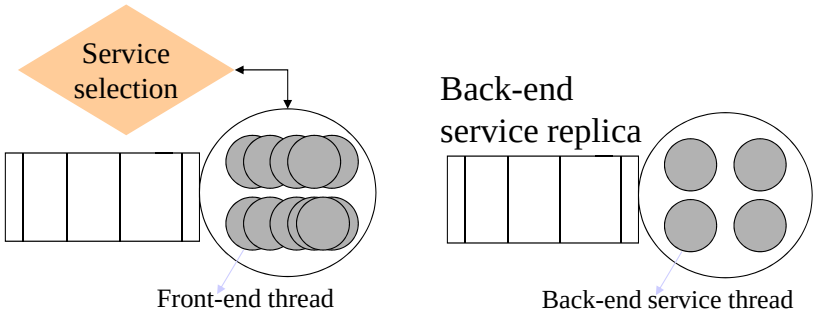


Figure 3.2. Queueing schematics for replicas

replication manager keeps at least one replica for each back-end service type, i.e., $n_{s_i}(t) \geq 1$, and for front-end replicas, i.e., $n_f(t) \geq 1$.

Front-end and back-end service replicas can be activated or deactivated in slotted windows by the replication manager. When a replica is deactivated, it receives no more requests (client requests in the case of a front-end replica, service invocations from a front-end replica in the case of a back-end service replica), but it still needs to complete the processing of all pending requests in its queue. We assume it takes constant warm-up time for a replica before starting to process the incoming requests.

3.1.5 Average End-to-end Request Response Time, R_a

The average end-to-end response time of a request for application a, R_a, is the summation of (1) the queue time at a front-end replica, Q_f, (2) the front-end execution times ($|\Omega_a| d_f$), and (3) the response time of all service invocations composed in Ω_a. Thus, one can obtain

$$R_a = Q_f + |\Omega_a| d_f + \sum_{i \in \Omega_a} R_{s_i}, \tag{3.1}$$

where $|\Omega_a|$ denotes the cardinality of Ω_a, i.e., the number of invocations in a service composition of application a. Herein, we assume the network time is negligible compared to the processing time and queueing time at front-end and back-end service replicas.

3.2 Optimizing Performance of Front-end and Back-end Service Replicas

System utilization has commonly been used as a performance metric for designing resource provisioning policies [Verma et al., 2007; Chen et al., 2005]. Typically, the target utilization is purposely kept below the maximum capacity, e.g. 80%, for handling variations in the workloads [Petrucci et al., 2011]. Certain load balancing algorithms can be very effective in reducing the variance of workloads and greatly enhance application response times [Björkqvist et al., 2011], given the same levels of resource provisioning and system utilization, especially when the system is moderately loaded. Combining both observations, we propose a hierarchical solution to attain a cost-performance effective service-oriented system: (1) coarse-grained front-end and back-end service replica provisioning by the replication manager, and (2) fine-grained load balancing algorithms among available service replicas.

3.2.1 Monitoring and Predicting Workloads

To design a replica provisioning policy, the very first step is to monitor and further predict the workloads [Singh et al., 2010; Chen et al., 2005; Petrucci et al., 2011]. As there are two distinct tiers in our system, namely front-end and back end, their workload characteristics need to be monitored separately. At the front-end tier, we focus on request rates of each application, whereas at the back-end tier we collect statistics of total invocations rates of each service.

We let λ_a be the request rate of application a. The total request rate received by the front-end tier is the summation of all applications, i.e., $\lambda = \sum_a \lambda_a$. We denote the invocation rate of requests received for back-end replicas of service i by λ_{s_i}. As an application request consists of various and multiple service invocations, the total request rate is less than the total service invocation rates, i.e., $\sum_a \lambda_a \ll \sum_i \lambda_{s_i}$. Note that λ and λ_a fluctuate in multiple time scales, and so does λ_{s_i}.

The replication manager monitors the request rates of all applications and invocation rates of all back-end services for all control windows. At the beginning of the control window, the replication manager obtains the estimate of $\lambda_a(t)$, and $\lambda_{s_i}(t)$, using historical statistics. In particular, a simple last value prediction is used, i.e., the arrival rate of the previous control window,

a	$a \in \{1 \dots A\}$ subscript for applications
i	$i \in \{1 \dots I\}$ subscript for services
Ω_a	sequence of service invocations for a request of application a
λ	total request rate
$\lambda_{\{a,s_i\}}$	request rate for application a and service i
$t_{\{f,s\}}$	number of threads in a front-end/back-end service replica
$n_{\{f,s_i\}}$	number of front-end/back-end i service replicas
$d_{\{f,s_i\}}$	average front-end/back-end i service execution time
R_{s_i}	average response time of back-end service i
Q_f	average queueing time of front-end
P	non-blocking probability for front-end threads
$U_{\{f,s_i\}}$	nominal utilization of front-end/back-end service i replicas
R_a	average end-to-end response time of requests from application a

Table 3.1. Notations and definition

$$\begin{aligned} \widehat{\lambda_a(t)} &= \lambda_a(t-1) \\ \widehat{\lambda_{s_i}(t)} &= \lambda_{s_i}(t-1). \end{aligned} \tag{3.2}$$

3.2.2 Controlling Replicas

Due to the blocking I/O in front-end threads, we consider two types of utilization the front-end tier: nominal and effective. The former computes the fraction of time front-end replica threads are busy processing compositions, whereas the latter computes the fraction of time front-end threads are busy or blocked waiting for back-end service invocation requests to return. For back-end service replicas, the effective and nominal utilization are equivalent. Particularly, the replication manager aims at maintaining the effective utilization of active front-end and back-end services replicas at the target values, U^*. In the following, we first derive the effective utilization and then obtain the replica control policy for back-end and front-end replicas, respectively.

Back-end Service Replicas

The utilization of active back-end service i replicas, U_{s_i} is defined as the invocation rate, λ_{s_i}, divided by the aggregate capacity provisioned, i.e., $n_{s_i} t_s / d_{s_i}$, according to the utilization law [Kleinrock, 1975]. At every control window, the

replication manager provides a sufficient number of replicas, $n_{s_i}(t)$, such that the effective utilization is less than the target value,

$$U_{s_i}(t) = \frac{\lambda_{s_i}(t) d_{s_i}}{n_{s_i}(t) t_s} \leq U^*, \forall i, t, \tag{3.3}$$

After substituting $\widehat{\lambda_{s_i}(t)}$ and following algebraic manipulation, the replication manager controls $n_{s_i}(t)$ by following

$$n_{s_i}(t) = \lceil \frac{\widehat{\lambda_{s_i}(t)} d_{s_i}}{U^* t_{s_i}} \rceil, \forall i, t. \tag{3.4}$$

Front-end Replicas

The effective utilization of front-end replicas considers the blocking I/O in dealing with sequential back-end service invocations. We let $P(t)$ be the non-blocking probability of sequential invocations within a composition. The effective capacity of all front-end replicas at window t is the product of the aggregate front-end capacity and non-blocking probability, i.e., $n_f(t) t_f / d_f P(t)$. The workload sent to front-end replicas from application a is the request rate multiplied by the number of invocations in a composition, $\lambda_a |\Omega|$. Therefore, the aggregate workload of the front-end replicas at window t is:

$$\lambda_f = \sum_a \lambda_a |\Omega_a|.$$

Similar to Equation 3.3, one can then write the effective utilization of front-end replicas at window t as

$$\begin{aligned} U_f^{eff}(t) &= \lambda_f(t) \frac{d_f}{n_f(t) t_f P(t)} \\ &= U_f(t) \frac{1}{P(t)} \leq U^*, \end{aligned} \tag{3.5}$$

where U_f denotes the nominal utilization. One can see that U_f is higher than U_f^{eff} by a factor of blocking probability, P.

We derive $P(t)$ as the weighted average of the non-blocking probability from applications, because the blocking depends on the composition defined in the application. We let $P_a(t)$ be the non-blocking probability of application a, then write

$$P(t) = \sum_a \frac{\lambda_a(t)}{\lambda(t)} P_a(t),$$

where $\frac{\lambda_a}{\lambda}$ is the percentage of application a requests out of total application requests.

The non-blocking probability of application a requests can be derived from the fraction of the front-end processing time over the summation of front-end processing and blocking time. For a composition request, the front-end processing time is the processing time per invocation multiplied by the number of invocations, $d_f|\Omega_a|$. The blocking time is essentially the summation of back-end service response times of all invocations, $\sum_{i\in\Omega_a}(R_{s_i})$. As such, we can express P_a as a function of d_f and R_{s_i},

$$P_a(t) = \frac{|\Omega_a|d_f}{\sum_{i\in\Omega_a} R_{s_i}(t) + |\Omega_a|d_f}. \tag{3.6}$$

Note that $R_{s_i}(t)$ here is not stationary as the provisioning of back-end service replicas changes across different time windows. To obtain $\widehat{P_a(t)}$, we propose to substitute $R_{s_i}(t)$ by an estimate based on last window statistics,

$$\widehat{R_{s_i}(t)} = R_{s_i}(t-1),\ \forall i, t. \tag{3.7}$$

Using the estimated total request rate, the application request rate, and the response time of back-end services, one can obtain

$$\begin{aligned}\widehat{P(t)} &= \sum_a \frac{\widehat{\lambda_a(t)}}{\widehat{\lambda(t)}}\widehat{P_a(t)} \\ &= \sum_a \frac{\widehat{\lambda_a(t)}}{\widehat{\lambda(t)}}\frac{|\Omega_a|d_f}{\sum_{i\in\Omega_a}\widehat{R_{s_i}(t)} + |\Omega_a|d_f}.\end{aligned} \tag{3.8}$$

Combining Equations 3.5 and 3.8 and using some algebraic manipulations, the replication manager controls the number of front-end replicas by the following:

$$\begin{aligned}n_f(t) &= \lceil \frac{\widehat{\lambda_f(t)}d_f}{\widehat{P(t)}t_f U^*} \rceil \\ &= \lceil \frac{\widehat{\lambda_f(t)}d_f}{t_f U^* \sum_a \frac{\widehat{\lambda_a(t)}}{\widehat{\lambda(t)}}\frac{|\Omega_a|d_f}{\sum_{i\in\Omega_a}\widehat{R_{s_i}(t)}+|\Omega_a|d_f}} \rceil,\ \forall t.\end{aligned} \tag{3.9}$$

In summary, the replication manager monitors statistics relating to application request rates, back-end service invocation rates, the utilization of front-end

and back-end service replicas, and the response time of back-end service invocations. Using the collected and estimated statistics, the replication manager activates and deactivates replicas at the beginning of each window. Note that the statistics monitored in the replication manager can easily be collected on production systems.

3.2.3 Bounding Analysis on Front-end Performance

One can see that the effective utilization of front-end replicas is higher than the nominal utilization, which is commonly measured by utility tools. Following our model and analysis in the previous subsection, we derive the upper bound of nominal utilization as a function of the target utilization values. Consequently, one can use such an upper bound as a simple rule of thumb for evaluating the performance of the front-end tier of service-oriented systems.

Theorem 3.2.1. *The upper bound of nominal utilization of front-end replicas is*

$$U_f \leq (U^*)^2.$$

The upper bound of U^c is achieved when the non-blocking probability is equivalent to the target utilization, $P = U^*$.

Proof. We start the proof by first deriving a looser upper bound of the nominal utilization. Then, using the optimal value of non-blocking probability, we can reach a tighter bound, which only depends on the target utilization.

From Equation 3.5, one can write $U_f \frac{1}{P} \leq U^* \leq 1$. First, as $P \leq 1$, we know $U_f \leq U^*$. Secondly, as $U_f \frac{1}{P} \leq 1$, we know $U_f \leq P$. Combining both observations, one can get a loose upper bound of U_f, by taking the minimum of P and U^*, $U_f \leq \min\{U^*, P\}$.

When $U^* \geq P$, $U_f \leq P$; whereas when $U^* \leq P$, $U_f \leq U^*$. Consequently, the upper bound of U_f increases in P and stays constant at U^*, after P reaches U^*. In other words, when $P = U^*$, the upper bound U_f is maximized. Taking $P = U^*$ into Equation 3.5, one can get $U_f \frac{1}{U^*} \leq U^*$, and then $U_f \leq (U^*)^2$. □

Theorem 3.2.1 points out that to achieve the nominal utilization upper bound, the non-blocking probability should be at least as high as the target utilization. However, the non-blocking probability at front-end replicas is bounded by the relative difference between the front-end processing time and response time of back-end service invocation. The maximum achievable non-blocking probability is when there is no queueing time at the back-end service replicas. Comparing such a non-blocking probability with the target utilization, one can gauge how

tight the nominal utilization is bounded by $(U^*)^2$. When the maximum achievable non-blocking probability is lower than U^*, the nominal utilization is lower than $(U^*)^2$. whereas when the maximal achievable non-blocking is greater than U^*, the nominal utilization might reach $(U^*)^2$.

3.3 Service Selection

To evenly balance the loads on the distributed replicas, we adopt two back-end service selection algorithms. For each service invocation in a request, a front-end thread selects back-end end service replicas using only statistics collected at the local front-end replica. That is, threads of a replica have the back-end service replica statistics from their local requests, but not the aggregate statistics from all front-end replicas. In the following, we describe two selection policies:

1. **Distributed Round Robin Selection (D-RR):**
   Each front-end replica maintains a round-robin list of active back-end service replicas. At the beginning of each control window, the list is updated by adding (removing) the newly activated(deactivated) back-end service replicas. Upon back-end service replica selection, the front-end replica thread requests the next back-end service replica from the round-robin list and sends the invocation request to the chosen back-end replica. D-RR is completely load oblivious and the resulting loads on back-end service replicas may not be optimally balanced.

2. **Distributed Shortest Queue Selection (D-SQ):**
   A front-end replica keeps statistics of outstanding service invocation requests sent by its threads and the corresponding queueing information at active back-end service replicas. Upon back-end service replica selection, the front-end thread selects the back-end replica with the lowest number of queued invocations based on the locally maintained statistics. The implementation overhead is limited compared to the conventional shortest queue selection, which collects queueing statistics from all front-end replicas. D-SQ is partially load aware, practical, and has good potential for reducing response time [Björkqvist et al., 2011] and balancing loads on back-end replicas.

As D-SQ is expected to achieve lower response times of service invocation than D-RR, one can expect that the resulting non-blocking probability is higher for D-SQ, according to Equation 3.8.

3.4 Evaluation

In this section, we evaluate our proposed replication policy in combination with two load balancing schemes using trace-driven simulation. We first describe the simulated environment: the workload generator and the system scenarios. Our evaluation results, based on the average of ten simulation runs, show that our proposed replication policy can effectively reduce the number of front-end and back-end services replicas, while maintaining the target utilization and minimizing the response time of back-end service invocations.

3.4.1 Simulator and System Configuration

We built an event-driven simulator of service-oriented systems in Java, as shown in Figures 3.1 and 3.2. Composition requests are generated from applications. A front-end replica has $t_f = 32$ threads to process service compositions and invocation in parallel. The execution time per front-end thread is assumed exponentially distributed with an average $d_f = 0.5s$. A back-end service replica is configured to have $t_s = 4$ threads, independent of service types. The replication manager collects workload statistics at every control window and activates/deactivates front-end and back-end replicas at the beginning of a window. The length of the control windows is chosen according to workload characteristics and prediction schemes.

Simulation Workload

The arrival patterns of requests from different applications are typically not available to the public, due to the business confidentiality. The most widely used traces are World Cup web site workloads that date back to 1998 [Arlitt and Jin, 2000; Petrucci et al., 2011], or are derived from the TPC-W benchmark [Singh et al., 2010], which was last updated in 2001. In contrast to conventional approaches, we seek an alternative to generate the workload – converting the CPU utilization traces of an existing production system into the workload input of a discrete simulator [Verma et al., 2007; Chen et al., 2005; Meng et al., 2010]. According to the basic utilization law [Kleinrock, 1975], the utilization multiplied by a normalized constant is essentially the request rate, especially when the load is below 100% utilization.

We collect utilization traces from four servers providing web services at financial, airline and media industries, during 10 am-12pm on October 20, 2011. The trace from one server is considered as one application. The utilization values are

the average computed over 15 minutes. To obtain the request rate per second, we multiply the utilization values with the processing power of the server, i.e., the number of cores. We illustrate the rationale by an example: Let the utilization value be 35% for a 16 core server. This implies that, on average, 5.6 ($0.35 \cdot 16$) cores are busy. We further assume that a core is occupied by a single request and such a value corresponds to the request arrival rate for a small granularity, i.e., second. As such, we obtain the request rates for four applications, shown in Figure 3.3. One can clearly see that the workloads are time-varying.

Due to the limitation of the coarse granularity in collecting utilization, we are unable to collect the higher moment statistics and further fit the empirical distribution of utilization. Consequently, we assume that the arrivals of requests follow Poisson processes for each 15 minutes and that their means fluctuate according to Figure 3.3. Once requests are generated, they are then immediately forwarded to available front-end replicas in a random fashion.

Simulated System Scenarios

In particular, we consider the following two specific system scenarios and their compositions:

- System scenario I:
  The system provides a single type of back-end service, namely S_0. Requests are generated from two applications, i.e., $a = \{1, 2\}$, whose requests rates correspond to app1 and app2 in Figure 3.3. Their service compositions are $\Omega_1 = \langle S_0, S_0 \rangle$, $\Omega_2 = \langle S_0, S_0, S_0 \rangle$. The execution time of a back-end service replica thread at S_0 is assumed exponentially distributed with mean $d_{s_0} = 1$. The maximum number of available front-end and back-end service replicas are $n_f = 9$, $n_{s_0} = 33$. The length of each control window is 100 seconds.

- System scenario II:
  The system provides three back-end service types, namely S_0, S_1 and S_2. Composition requests are generated from four applications, whose requests rates correspond to app1, app2, app3 and app4 in Figure 3.3. Their service compositions are $\Omega_1 = \langle S_0, S_1, S_0 \rangle$, $\Omega_2 = \langle S_0, S_2 \rangle$, $\Omega_3 = \langle S_0, S_1, S_2 \rangle$, and $\Omega_4 = \langle S_2, S_0, S_1 \rangle$. The execution times of a back-end service replica thread at S_0, S_1 and S_2 are assumed exponentially distributed with means $d_{s_0} = 1$, $d_{s_1} = 1.5$, and $d_{s_2} = 2.5$ seconds, respectively. The maximal number of available front-end replicas and S_0, S_1 and S_2 replicas are $n_f = 18$, $n_{s_0} = 19$, $n_{s_1} = 26$, and $n_{s_3} = 37$, respectively. The length of each control window is 150 seconds.

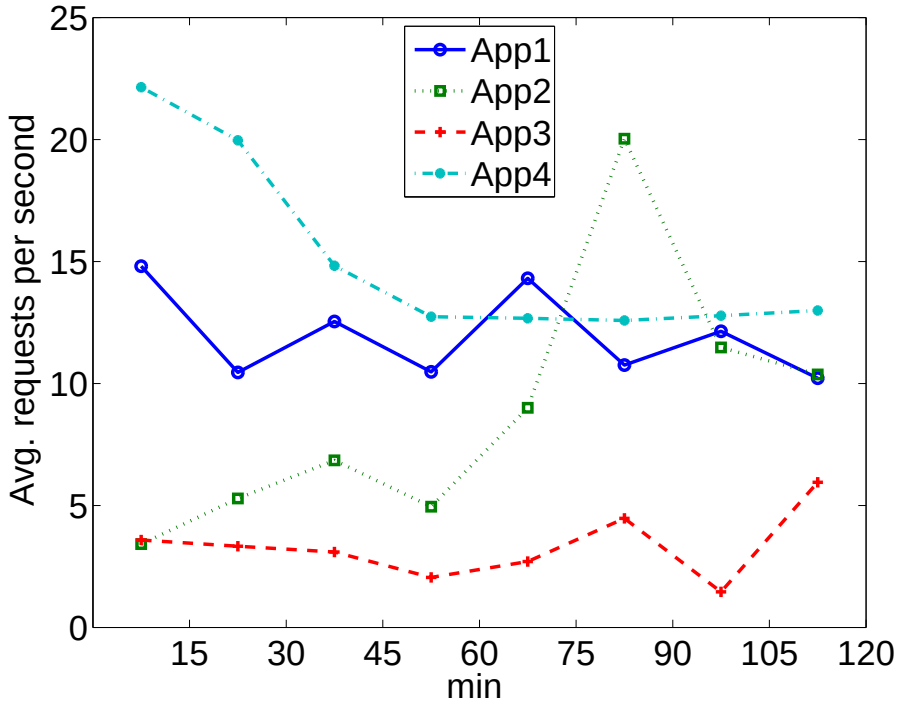


Figure 3.3. Request rates of applications, λ_a.

For both scenarios, we set the target utilization of the active front-end and back-end service replicas to be 85% and 80%, respectively. Such values are chosen by empirical experiences [Petrucci et al., 2011]. Note that our replication policy aims to maintain the front-end effective utilization, which includes the blocking time, at the target value. For each simulation run, we collect the performance metrics, averaged over all control windows, i.e., replica savings, nominal and effective utilization of front-end and back-end service replicas (U_f^{eff}, U_f, U_s), queueing time at front-end replicas (Q_f), response time of back-end service invocations (R_{s_i}), and end-to-end request response time (R_a). In particular, the replica savings are computed as one minus the number of total active replicas divided by the maximal number of available front-end and back-end service replicas. Using this metric, one can estimate the cost savings, given the target performance. For both system scenarios, we compute the average of the aforementioned metrics over ten simulation runs and present them in Tables 3.2, 3.3 and 3.4. Moreover, for the purpose of comparison, we additionally simulate a static replication policy which keeps the number of active back-end service replicas at the maximum for all control windows, independent of workloads. The lowest end-to-end response time can be achieved via maximum replica provisioning.

3.4.2 System Scenario I

We apply our replication policy on system scenario I, with two different workload load prediction schemes and two service selection schemes, D-SQ and D-RR, and summarize the performance metrics in Table 3.2. To verify the accuracy of workload prediction in our proposed replication policy, we use our replication policy with actual application request and invocation rates, and the default last value predictions. One can observe the performance degradation is $15-25\%$ when using last value prediction with our replication policy, with any given service selection.

When comparing D-SQ and D-RR under "actual" prediction, D-RR achieves similar replica savings and effective utilization as D-SQ; however, D-RR has roughly 20% higher invocation and end-to-end response time. The utilization of back-end service replicas is slightly under the target value of 80%, whereas the effective utilization of front-end replicas is roughly 15% lower than the target value, due to the large number of threads in a front-end replica. As expected, D-SQ can achieve a lower response time via better load balancing, and thus a lower non-blocking probability at front-end threads that is reflected by the relative difference between U_f^{eff} and U_f. Moreover, due to a low non-blocking probability, the front-end nominal utilization is way lower than the its upper bound, according to Theorem 3.2.1 one can expect D-RR to have even worse performance when the workload prediction is inaccurate, i.e., over- and under-estimating. Consequently, we provide a higher spare capacity for the front-end tier and set a slightly lower utilization target when applying our replication policy with last value prediction and D-RR, i.e., 80% and 75%, respectively, for both scenario I and II.

When applying our proposed replication policy with the last value prediction specified in Equation 3.2, the replica savings are around 50%, and D-RR has slightly lower replica savings due to a lower target utilization. The average queuing time at front-end replicas is significantly higher than in the "actual" case, and consequently the end-to-end response time of the applications is higher than in the "actual" case by $15-25\%$. Even with higher provisioning of front-end and back-end service replicas, our proposed replication policy with last value prediction and D-RR selection still has the worst queueing time at the front-end tier and consequently the worst application end-to-end response times. As pointed out earlier, the back-end service selection can fine tune the performance, but the provisioning of the replicas are the first order parameters to control.

Overall, our proposed replication policy together with last value prediction can achieve (1) significant replica savings; (2) front-end and back-end service utilization that is slightly under the target values; and (3) very low end-to-end

Table 3.2. Performance of applying proposed replication policy on system scenario I.

Proposed replication policy									
Workload		Performance Statistics							
Load Prediction	Service Selection	Replica Savings [%]	U_f^{eff} [%]	U_f [%]	U_{s_0} [%]	$Q_f[s]$	$R_{s_0}[s]$	$R_{a_0}[s]$	$R_{a_1}[s]$
Actual	D-SQ	50.00	69.85	22.12	75.55	0.03	1.08	3.20	4.77
Actual	D-RR	50.00	72.80	20.71	75.55	0.06	1.26	3.57	5.33
Last value	D-SQ	50.68	70.93	21.87	75.91	0.75	1.12	4.00	5.63
Last value	D-RR	47.53	69.96	20.08	71.31	1.10	1.25	4.60	6.34
Maximum Static Provisioning of Replicas									
none	D-SQ	0.00	26.71	8.78	38.33	0.00	1.02	3.03	4.55
none	D-RR	0.00	26.81	8.78	38.33	0.00	1.02	3.04	4.56

request response times that are only slightly higher than the response times under static maximum provisioning.

3.4.3 System Scenario II

We summarize the performance metrics of applying our proposed replication policy with "actual" and "last value" prediction in Tables 3.3 and 3.4. Following the observation and rationale in scenario I, we set the target utilization of front-end and back-end service replicas to 80% and 75%, respectively.

One can make the following general observations, which are similar to the ones made in scenario I: The replica savings achieved by our proposed replication policy are quite significant, compared to providing the maximum number of replicas in all windows. Our proposed replication policy maintains the front-end and back-end service replica utilization just slightly below the target values. In particular, when applying our proposed replication policy with "actual" prediction, the average end-to-end response time, R_{a_0}, R_{a_1}, R_{a_2} and R_{a_3}, is roughly 10% higher than with static maximum replica provisioning. It strongly supports the accuracy of our proposed replication policy in predicting performance metrics, especially in a more complex system. The difference between "actual" and "last value" prediction is more visible in front-end queueing time (Q_f) and thus degrades the end-to-end response time roughly by $10-20\%$.

We plot the run time results of applying our proposed replication policy with last value prediction and D-SQ in Figure 3.4. The number of front-end and back-end service replicas is highly correlated, because the number of front-end replicas determines the invocation rates received by back-end service replicas. As such, the utilization of front-end replicas oscillates in a greater range than the back-end service utilization. Queueing time at the front-end replicas is fairly low, except for two spikes around 70 and 80 minutes. The invocation response times for all back-end services are even more stable, except for a spike around 80 minutes.

Table 3.3. Replica savings and replica utilization when applying our proposed replication policy on system scenario II.

Proposed replication policy							
Workload		Performance Statistics					
Prediction	Selection	Savings [%]	U_f^{eff} [%]	U_f [%]	U_{s_0} [%]	U_{s_1} [%]	U_{s_2} [%]
Actual	D-SQ	36.59	81.37	17.50	76.16	76.90	76.83
Actual	D-RR	36.59	82.54	16.29	76.16	76.90	76.84
Last value	D-SQ	36.63	81.77	17.34	75.65	76.25	76.49
Last value	D-RR	32.59	77.77	15.85	70.95	71.53	72.08
Maximum Static Provisioning of Replicas							
none	D-SQ	0.00	38.56	8.89	51.62	51.82	46.07
none	D-RR	0.00	38.59	8.89	51.62	51.82	46.07

Table 3.4. Front-end queueing time, and service and application response times, when applying our proposed replication policy on system scenario II.

Proposed replication policy									
Workload		Performance Statistics							
Prediction	Selection	$Q_f[s]$	$R_{s_0}[s]$	$R_{s_1}[s]$	$R_{s_2}[s]$	$R_{a_0}[s]$	$R_{a_1}[s]$	$R_{a_2}[s]$	$R_{a_3}[s]$
Actual	D-SQ	0.17	1.16	1.74	2.90	4.07	7.47	5.23	7.47
Actual	D-RR	0.28	1.29	1.94	3.23	4.52	8.24	5.80	8.24
Last value	D-SQ	1.47	1.18	1.76	2.97	5.41	8.89	6.62	8.88
Last value	D-RR	1.38	1.23	1.84	3.15	5.46	9.11	6.76	9.10
Maximum Static Provisioning of Replicas									
none	D-SQ	0.00	1.06	1.60	2.61	3.66	6.77	4.68	6.77
none	D-RR	0.00	1.06	1.60	2.61	3.66	6.78	4.67	6.77

Overall, our proposed replication policy is able to provide sufficient numbers of front-end and back-end service replicas, keep them well utilized, and maintain stable response times, given the load fluctuation over time.

3.5 Assumptions and Limitations

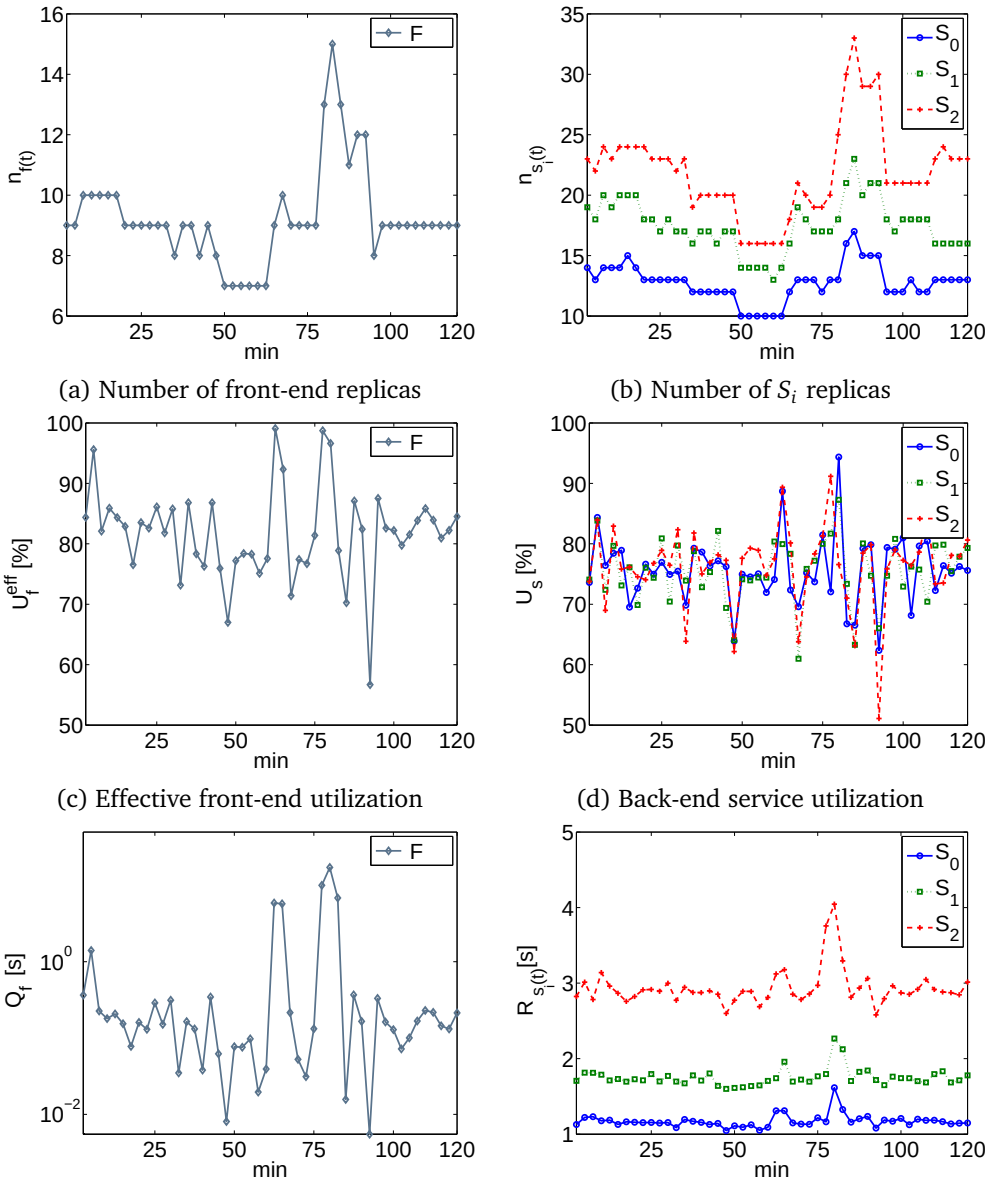
In this work, we make a number of assumptions to facilitate the development of a system model, enabling the analysis, as well as simplifying the simulations. From the service point of view, we assume single resource bound (in particular, CPU-bound) services. We also only consider composed services where the individual back-end services are invoked in a sequential manner. All components are considered to be stateless in the sense that new replicas can be created, and existing replicas terminated, without needing to transfer any state to other entities in the system. On the other hand, the front-end service replicas do keep state for each invocation of a composed service, to be able to determine which back-end responses belong to which composed service request, and to be

able to send responses back to the client when the composed service request has been completed. We also do not consider the network in terms of e.g., latency or throughput, in our analysis or evaluation. For the request processing we assume first-come-first-served both on the front- and back-end service replicas, and we evaluate our approach using (distributed) join-the-shortest-queue and round robin as the load balancing policies.

The aforementioned assumptions make up some significant limitations that need to be taken into account when applying the results on real systems, or when extending the modeling and simulation work to more complex scenarios. Parallel executions of back-end services to model more complex composed services, maintaining state and consistency between replicas, and more accurate models of resource constraints are all viable avenues to explore for future work.

3.6 Summary

In this chapter, we studied a service-oriented system hosting multiple front-end and back-end service replicas. Our system model captures the workload dynamics and the interdependency between the front-end and back-end service replicas equipped with multiple threads. To reduce operational cost, as well as minimize the end-to-end response time of applications, we developed a dynamic replication manager. The replication manager periodically adjusts the provisioning of replicas such that the effective utilization of both front-end and back-end replicas is kept at target values. The replication manager explicitly factors in the dependency between the front-end and back-end tiers, using the derivation of non-blocking probability at front-end replicas. Furthermore, we provide theoretical bounding analysis on front-end replicas and derive optimal/maximal nominal utilization. Our trace-driven simulation results show that using our proposed replication policy, along with simple last-value workload prediction, can achieve great replica savings and keep front-end and back-end service replicas well utilized, while maintaining low response times, especially when the loads on back-end service replicas are well balanced.



(a) Number of front-end replicas

(b) Number of S_i replicas

(c) Effective front-end utilization

(d) Back-end service utilization

(e) Front-end queueing time

(f) Response time of back-end service invocation

Figure 3.4. Run time control and performance of proposed replication policy with D-JQ, on scenario II.

Chapter 4

Leveraging Performance Variability for Service Provisioning

There is an emerging trend to deploy services in cloud environments due to their flexibility in providing virtual capacity, ease of management, and pay-as-you-go billing features. Cost-aware services demand computation capacity such as virtual machines (VMs) from a cloud operator according to the workload (i.e., service invocations) and pay for the amount of capacity used according to billing contracts. However, as recent empirical studies show, the performance variability, i.e., non-uniform VM performance, is inherently higher than in private hosting platforms This can be explained by the fact that the cloud operators may consolidate VMs of multiple tenants on the same physical machine, resulting in resource sharing and possible performance interference. Additionally, the cloud providers typically run their cloud infrastructure on top of heterogeneous hardware, which can also cause performance variability. Consequently, the provisioning of service capacity in a cloud needs to consider varying VM performance as well as workload variability.

In this chapter we develop an opportunistic replication policy for elastic service provisioning on cloud platforms. Our objective is to leverage the variability in VM performance and their billing contracts in a cloud such that the VM costs of all services hosted by a provider are minimized, while maintaining given system utilization. Our policy takes several control actions in a slotted window: turning VMs on and off, replacing slower VMs in the hope of getting faster ones, and reconfiguring VMs from one type of service to another. All these actions are associated with non-negligible time overhead. The criteria are the predicted workload, estimated VM performance, target system utilization, and billing contract periods. Our evaluation results based on simulation show that the proposed

opportunistic replication policy achieves significantly lower service provisioning costs than workload-oblivious or purely workload-driven policies.

In this work, we analyze the workload of incoming requests of *multiple service types*, where the request can be satisfied by one of many replicated service replicas. We only consider *stateless services*, and assume that new replicas, identical with existing ones, can be started up at any time, and with only a short delay. When deciding on how many resources are required to provision the provided services, we assume CPU-bound services, look at the *utilization* of each service replica, and compare it to a target utilization. We use the utilization as the performance measure instead of e.g., the response time or the throughput, since it can be easily obtained by a monitoring tool without having to modify and instrument the software providing the actual service.

The original scientific contribution of the work presented in this chapter is a novel service replication policy, which is specially designed to explore the variability of VM performance on cloud platforms. In contrast to existing replication polices, we optimize the cost and performance not only for a single service but also for the entire system, by an augmented set of control actions, in particular replacing and reconfiguring VMs. Our evaluation environment encompasses a large number of different parameters, such as different time overheads associated with each control action. The proposed opportunistic replication policy is shown to achieve lower cost and better performance for services hosted in the cloud, compared to replication policies oblivious to the unique characteristics in the cloud.

The rest of this chapter is organized as follows: The system architecture is explained in Section 4.1. The proposed opportunistic replication policy in Section 4.2. Section 4.3 contains the experimental results. Section 4.4 lists the assumptions and limitations of our work, and Section 4.5 summarizes the chapter.

4.1 System Model

4.1.1 System Architecture and Dynamics

Figure 4.1 illustrates the system architecture considered in this chapter. A service provider deploys I types of services S_i ($1 \leq i \leq I$) in a cloud. The services considered here are simple atomic ones (i.e., we do not focus on composite services that invoke other services). At any given moment, there are $n_i \geq 1$ VMs running a service of type i; we also say there are n_i replicas of service i in the cloud. The values n_i may change over time according to the actions taken by the

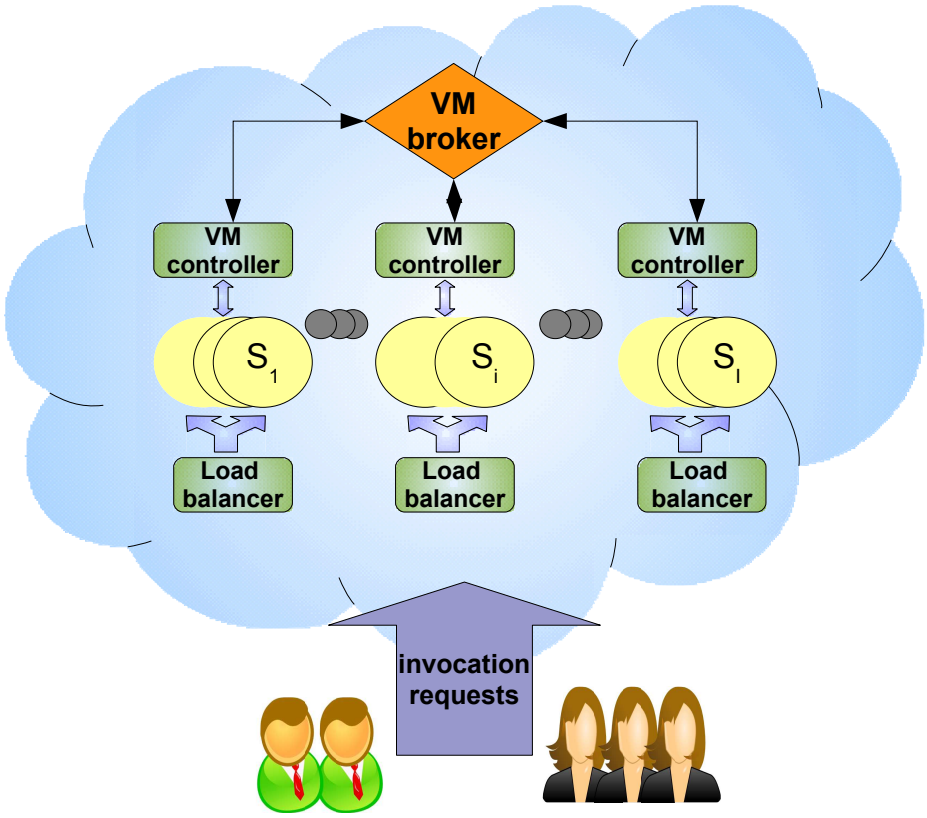


Figure 4.1. Schematics of services system deployed in a Cloud platform.

policy presented in this chapter. However, there is always at least one replica for each service.

To limit the scope of this study, we assume that all services are CPU-bound and multi-threaded, that is, capable of handling several concurrent invocation requests in parallel. We also assume that the service execution time is not significantly influenced by the input parameters passed upon service invocation.

For each service, there is a corresponding load balancer and VM controller that are also deployed in the cloud. The load balancer distributes incoming invocation requests to the replicas of the requested service (i.e., to the currently active VMs running a service of the corresponding type) with the fewest outstanding requests. We assume that the size of invocation requests varies, following an exponential distribution, and thus the execution times of requests follow an ex-

ponential distribution for a given VM throughput. The VM controller monitors active VMs and keeps tracks of statistics about the invocation rate, VM performance, and the billing periods of the active VMs. All controllers communicate the statistics to the VM broker, on which the proposed opportunistic policy and the control actions are implemented.

The throughput of a VM (i.e., its performance) is not fixed, but changes over time due to the possibly heterogeneous infrastructure used by the cloud operator, hardware optimizations that result in performance fluctuations, performance interference of multiple VMs consolidated on the same physical machine, and VM migrations. In this chapter, we assume that the average performance of VMs with the same specification fluctuates in the discrete range of values. The specific values of VM performance can be estimated by observing the completed service requests. Each VM is bound to a contract that defines the billing period (e.g., one or more hours). Therefore, releasing a VM before the end of a billing period would be wasteful for the service provider who would still have to pay until the end of the period.

4.1.2 VM Replica Provisioning

Here, the VM replication provisioning is implemented in slotted windows. The length of the control windows depends on the dynamics of the workload and the parameterization of the service replication policy. We assume that the billing period is a multiple of the algorithm's execution interval. In our simulation, we use a billing period of one hour. To dynamically provide VM replicas in a cost-effective manner, the VM broker considers four kinds of control actions: (1) turn on a new VM; (2) turn off a VM (i.e., terminating the contract at the end of a billing period); (3) replace a VM at the end of a billing period, if the VM is suspected of not performing well; (4) reconfigure a (previously allocated) VM to run a service of a different type. The first three actions are requests towards the cloud operator, while the fourth action is transparent to the cloud operator.

There are some time overheads associated with each action. The turning on of a new VM is assumed to take υ seconds to load and start the required service. The VMs that are about to be turned off need to immediately stop receiving invocation requests, but they will complete serving any requests that are currently being processed, or are in the queue waiting to be processed. As for VMs reconfigured from one service to another, they no longer receive invocations of the former service and start serving invocations of the new service right after completing the remaining requests of the former service and after the reconfiguration process. Here, similar to the process of loading services, we assume that the reconfig-

uration also takes υ seconds. The newly turned-on and reconfigured VMs are published as "available" VMs after the completion of their loading/configuration process. Note that previous related studies [Chen et al., 2005; Lin et al., 2013; Singh et al., 2010; Stewart et al., 2007] tend to overlook the overhead structure and lead to a simplified replication policy.

4.2 Opportunistic Replication Policy

Following the rule of thumb practiced in today's resource management [Verma et al., 2007; Chen et al., 2005], we provide sufficient VMs to each service such that the VMs' aggregate capacities are well utilized. We use a typical target utilization of around 80% [Petrucci et al., 2011], for handling temporary workload variation. Here, we aim at achieving better performance metrics, e.g., response time, and maintain target utilization at a lower cost, by leveraging a pay-as-you-go billing model and the variability in VMs' performance in the cloud.

We develop an opportunistic replication policy and implement it in the VM broker. In contrast to replication policies on private platforms, our proposed policy decides not only on the number of active VMs per service, but also strives to acquire VMs with better performance. The general idea of our opportunistic policy is that the VM broker first decides on the number of VMs for each service, based on the information monitored/collected in VM controllers. The second step is to select specific VMs, using appropriate control actions. The selection criteria considered are the billing period, the difference in the number of VMs in adjacent windows, and the performance of active VMs. Each controller then executes the decisions made by the broker. In the following, we first describe the control timing of the broker, the algorithm for deciding on the number of VMs for each service, and finally, the algorithm for selecting VMs.

4.2.1 Control Window and Overhead

Herein, we consider a sequence $[T_t : 0 \leq t < k]$ of k windows, each of length τ minutes. Due to the time overhead associated with each control actions, the VM broker queries the required statistics from controllers at ϵ seconds before the beginning of every control window. The schematics are depicted in Figure 4.2. Controllers of services immediately send back their invocation rate, VM performance, and their billing periods. Using the collected information and algorithms described in the following two subsections, the broker computes and broadcasts its decisions of VM replication and selection to all controllers. We assume that

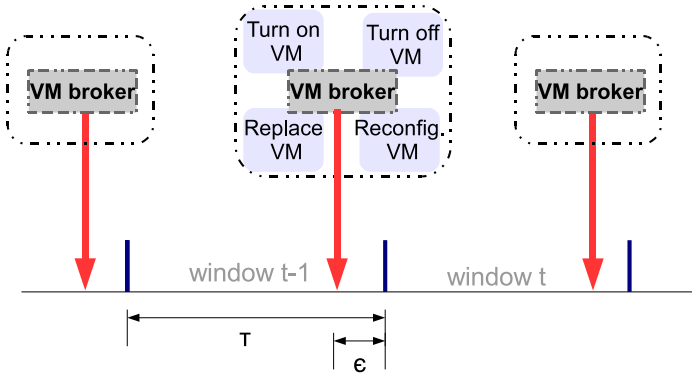


Figure 4.2. Timing of control actions and windows for the VM broker.

such a decision process takes a negligible time and no time overhead occurs. We choose such a value of ϵ that there is sufficient time for turned-on VMs to load the services, turned-off VMs to complete the remaining service invocations, replaced VMs to complete pending requests and replacement VMs to load the new service, and reconfigured VMs to complete pending requests and load the new service.

4.2.2 Number of Replica VMs

To proactively provide a sufficient number of well-utilized VMs at the beginning of every control window, the broker needs to know the utilization of active service VMs by the estimates of the average invocation rates, and the aggregate capacity.

We define the utilization of active service i VMs, U_i, as the invocation rate, λ_i, divided by the aggregate throughput of active VMs, i.e., $\sum_{j=1}^{n_i} \mu_{ij}$, where μ_{ij} denotes the throughput of VM j for service i,

$$U_i = \frac{\lambda_i}{\sum_{j=1}^{n_i} \mu_{ij}}. \tag{4.1}$$

Furthermore, we define μ_i as the average throughput per active VM, i.e., $\sum_{j=1}^{n_i} \mu_{ij} = n_i \mu_i$. Note that as there can be multiple threads in a replica VM, the performance of a VM corresponds to the summation of the performance of all of the threads. Our objective for the VM provisioning is that the effective utilization of every service in every window is less than the target value, $U_i(t) < U^*, \forall i, t$. To that end, we first need to estimate the average invocation rate and average capacity for

each coming control window. We propose to use a simple last value prediction for the invocation rate,

$$\widehat{\lambda_i(t)} = \lambda_i(t-1), \tag{4.2}$$

and for the average throughput,

$$\widehat{\mu_i(t)} = \mu_i(t-1) = \frac{\sum_{j=1}^{n_i(t-1)} \mu_{ij}(t-1)}{n_i(t-1)}. \tag{4.3}$$

Substituting the estimated values of Equation 4.2 and 4.3 into Equation 4.1, the broker estimates the utilization of service i when deploying $n_i(t)$ VMs at the beginning of window t,

$$U_i(t) = \frac{\widehat{\lambda_i(t)}}{n_i(t)\widehat{\mu_i(t)}}. \tag{4.4}$$

After straightforward algebraic manipulation, the broker controls $n_i(t)$ such that $U_i(t) \leq U^*$, as follows,

$$n_i(t) = \lceil \frac{\widehat{\lambda_i(t)}}{U^*\widehat{\mu_i(t)}} \rceil, \forall i, t. \tag{4.5}$$

Once the broker obtains the values of $n_i(t)$, it proceeds to decided on which VMs are to be turned off, replaced, and reconfigured, and how many new VMs are to be turned on.

4.2.3 Turning on-off, Replacing, and Reconfiguring VMs

The objective of selecting VMs is to maintain as few VMs as possible and to keep as many fast VMs as possible, such that the return on payment for active VMs is maximized. Consequently, the broker only turns off the expiring VMs, whose billing contracts end, and only turns on new VMs for services when there is no spare capacity from other services. In general, the broker is greedy in maximizing the "benefit" of individual services, rather than the global welfare of all services, when it comes to decreasing VMs. The broker also increases VMs in a collaborative manner, as elaborated in the following.

The decision process of the broker is structured into two parts: The first part focuses on the services which need to reduce VMs, and the second part focuses on services which need to increase VMs, from their current provision. Critical parameters considered are the number of expiring contracts, $E_i(t)$, the difference in the number of VMs in adjacent windows, $\delta_i(t) = n_i(t) - n_i(t-1)$, and the performance of the VMs. Expiring VMs can be either turned off, reconfigured

to other services, or replaced by other VMs, whereas non-expiring VMs can only be reconfigured. When $\delta_i(t) > 0$, the service i demands more VMs for window t, whereas when $\delta_i(t) < 0$, the service i tries to reduce the number of VMs by turning off or reconfiguring whenever possible. This implies that not all services can always reduce VMs as the workload decreases as shown in Equation 4.5, due to the billing periods and no other services requiring more VMs.

To facilitate selecting control actions for increasing and decreasing VMs, we keep two lists, an expiring list and a reconfiguration list, which record expiring and reconfigurable VMs, respectively. The lists are filled up during the "decreasing" part of the policy, and flushed out during the "increasing" part of the policy. Both lists are maintained in a slowest-first manner. For example, the broker always selects the slowest expiring VMs first into the expiring list, and distributes the slowest VMs first to the services with $\delta_i(t) > 0$.

Decreasing VMs

For services with $\delta_i(t) < 0$, the broker greedily optimizes the aggregate VM capacity of individual services by turning off expiring VMs or replacing the slow expiring VMs with faster ones.

When there are more expiring VMs than reduced VMs, $E_i(t) > |\delta_i(t)|$,, where $|*|$ denotes the absolute value and $|\delta_i|$ is the number of VMs needs to be reduced. the broker first turns off $|\delta_i(t)|$ expiring and slowest VMs. Then, the broker tries to replace remaining expiring VMs by comparing the corresponding cost and benefit. The cost of replacing VMs is the unavailability of its capacity during the time a replacement VM is being configured. The potential benefit is the chance of obtaining a VM with better performance. We thus derive the quantitative cost of replacing a VM j of service i as

$$C_{ij} = \epsilon \mu_{ij}. \tag{4.6}$$

Correspondingly, we derive the benefit of replacing an expiring VM j of service i as the expected capacity gain, which sums the product of probabilities of throughput levels, the throughput differences, and the control window length. Assuming a VM has K different levels of throughput and the probability of receiving a VM with throughput level k is P_k, one can write

$$B_{ij} = \sum_{k \neq j}^{K} P_k(\mu_{ik} - \mu_{ij})\tau. \tag{4.7}$$

When the benefit of replacing VM j of service i is greater than the cost, $B_{ij} > C_{ij}$, the broker replaces VM j by a new VM. Note that the replacing decision may not

necessarily lead to a faster VM. From Equation 4.6, one can see that probabilistically speaking, it is beneficial to replace VMs especially when currently expiring VMs are slow and the control window is longer.

When there is not a sufficient number of expiring VMs to be reduced, i.e., $E_i(t) < |\delta_i(t)|$, the broker first turns off $E_i(t)$ expiring VMs. Then, among the non-expiring VMs, it chooses the $\{|\delta_i(t)| - E_i(t)\}$ slowest VMs and adds them to the reconfiguration list, Ψ. Such a list is first filled up by services with $\delta_i(t) < 0$ in a sequential order of service index, and then flushed out by services with $\delta_i(t) > 0$ in a round robin fashion for reasons of fairness. As such, the time overheads associated with replacing and reconfiguring VMs can be minimized.

Increasing VMs

Once the broker completes the process of increasing VMs, it proceeds to the services requiring additional VMs. The broker first tries to distribute any available VMs on the reconfiguration list, and then turn on new VMs where required. Let the total number of VMs on the reconfiguration list be $n^{\Psi}(t)$, and total number of additional VMs from services with $\delta_i(t) > 0$ be $\Delta = \sum_{i \in \{\delta_i(t) > 0\}} \delta_i(t)$. When $n^{\Psi}(t) > \Delta$, it implies a sufficient number of VMs can be reconfigured and then distributed to other services. The broker distributes the slowest Δ VMs on the reconfiguration list to services with $\delta_i(t) > 0$ in a round-robin fashion. Thereafter, the remaining $\{n^{\Psi}(t) - \Delta\}$ VMs on the reconfiguration list are returned to their original services. Alternatively, when there is not a sufficient number of VMs on the reconfiguration list to meet the requirement for an increasing number of VMs, the broker only distributes $n^{\Psi}(t)$ VMs in a round-robin fashion, and turns on additional VMs according to unfulfilled demands. We summarize the opportunistic replication policy implemented on the broker in Algorithm 1.

4.3 Evaluation

In this section, we use trace driven simulation to evaluate the proposed opportunistic replication policy for service systems deployed in the cloud. The performance metrics evaluated are the VM costs, the average normalized throughput of VMs, the average utilization of active VMs, and the average response time of an invocation. To show the effectiveness of the VM broker, we also present the detailed statistics of control actions. We benchmark our policy against a workload oblivious replication policy and a purely workload driven policy. Our evaluation results, based on the average of ten simulation runs, show that the VM broker

Algorithm 1 Opportunistic Replication Policy of the Broker

Compute $n_i(t)$ as in Equation4.5 and $\delta_i = n_i(t) - n_i(t-1)$, $\forall\, i$.
for $i = 1$ to I services, with $\delta_i(t) \leq 0$ **do**
  if $E_i(t) > |\delta_i|$ **then**
    Turn off $|\delta_i|$ expiring VMs
    Replace up to $\{E_i(t) - |\delta_i(t)|\}$ servers based on Equation 4.7, and 4.6.
  else
    Remove $E_i(t)$ expiring VMs
    Add **slowest** $\{|\delta_i(t)| - E_i\}$ VMs to the reconfiguration list, Ψ.
  end if
end for
for For services with $\delta_i(t) > 0$ **do**
  if $n^{\Psi}(t) > \sum_i \delta_i(t) = \Delta$ **then**
    Distribute Δ VMs on the reconfiguration list round-robinly
    Return remaining $\{n^{\Psi}(t) - \Delta\}$ VMs back to original services
  else
    Distribute $n^{\Psi}(t)$ VMs on the reconfiguration list in round-robin
    Add $\{\Delta - n^{\Psi}(t)\}$ servers and distribute to the corresponding services in round-robin
    Empty the reconfiguration list
  end if
end for

can significant reduce the cost by acquiring a smaller number and faster VMs in a collaborative manner, while maintaining the system utilization slightly lower than the target values, $U^* = 80\%$, and achieving low average response times of invocations.

4.3.1 System Configuration

We built a trace-driven simulator of service-oriented systems in the cloud using Java. Invocation requests are generated for each service, following a Poisson process with time varying arrival rates. Each VM replica is configured to have one thread, independent of service types. Moreover, we assume a VM can have three different levels of performance to process requests of each service in our simulated cloud environment. To ease the analysis, we express VM throughput as a multiplier of the minimum throughput of each service, $\alpha\mu_i$. The specific values are $\alpha = \{1, 1.2, 1.5\}$, i.e., a VM has an average throughput of μ_i, $1.2\mu_i$ and $1.5\mu_i$ for processing request of service i. The probabilities of obtaining VMs with different α values are 0.5, 0.3, 0.2 respectively. The aforementioned values can be configured according to values measured in different cloud platforms.

The VM controller collects the required statistics $\epsilon = 20$ seconds before every control window of length $\tau = 5$ minutes and the VM broker immediately computes and implements the control actions, some of which have time overhead of $\upsilon = 20$ seconds. For a fair comparison, the timing of control actions in policy II are synchronized with the broker. The specific length of the control window

is chosen according to workload characteristics and prediction schemes. The discussion of the optimality of those values is beyond the scope of this chapter.

Cost Calculation

We follow the convention in today's commercial cloud [EC2, 2014] and use an hour as the billing period. The actual cost per billing period is different between cloud providers, and also varies depending on the requested VM specifications. We present our results in terms of relative cost savings, and the results are therefore not bound to any cloud provider in particular.

The total cost is the summation of all requested VM hours. When VMs are turned off before the end of billing period, they still need to pay for the remaining minutes. We add the cost for any possibly remaining periods immediately onto the windows when those VM are turned off.

4.3.2 The Workloads: Invocation Requests

Following approaches used in [Verma et al., 2007; Chen et al., 2005; Meng et al., 2010], we adopt the utilization traces from current IBM production systems as workload input for each service, i.e., to generate the invocation requests. Based on the basic utilization law [Kleinrock, 1975], the utilization multiplied by a normalized constant reflects the request rate, especially when the load is below 100% utilization.

We collect utilization traces from four large multi-processor servers engaging in web services in financial, airline and media industries, in late January, 2012. The trace from one server is considered as one service here. The utilization values are the average computed over 15 minutes. To obtain the request rate per second, we multiply the utilization values with the processing power of the server, i.e., the number of cores. We illustrate the rationale by an example: Let the utilization value be 35% for a 16 core server. This implies that, on average, 5.6 ($0.35 \cdot 16$) cores are busy. We further assume that a core is occupied by a single request and such a value corresponds to the request arrival rate for a small granularity, i.e., second. As such, we obtain the request rates for four services, shown in Figure 4.3. One can clearly see that the workloads are time-varying.

The execution times of each service follow the exponential distribution with mean $\frac{1}{\mu_1} = \frac{1}{1}, \frac{1}{\mu_2} = \frac{1}{1}, \frac{1}{\mu_3} = \frac{1}{10}, \frac{1}{\mu_4} = \frac{1}{8}$ seconds respectively. Note that due to the performance variability of VMs, the execution times can be scaled down by the multiplying factor, α, by 1.2 or 1.5.

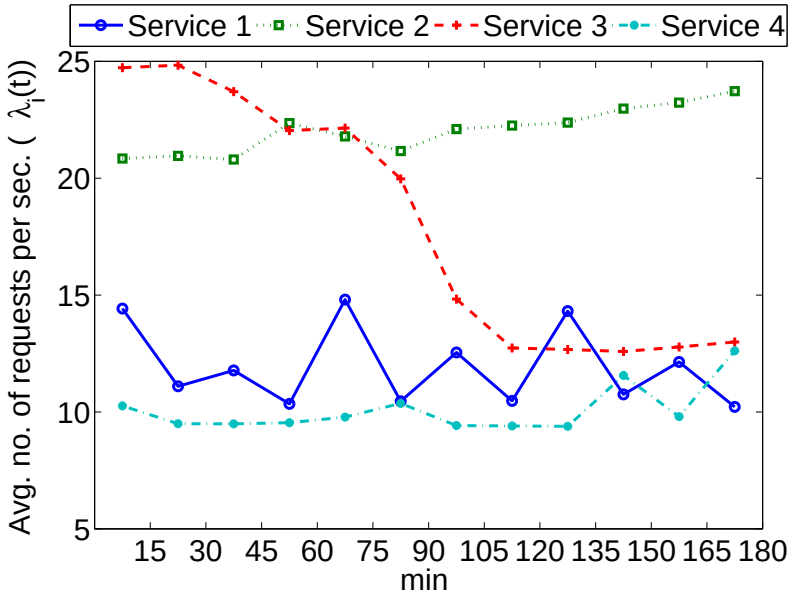


Figure 4.3. Average request rates of services, $\lambda_i(t)$.

Due to limitations in the granularity in collecting utilization we are unable to collect the higher moment statistics and further fit the empirical distribution of utilization. Consequently, we assume that the arrivals of the requests follow Poisson processes for each 15 minutes and that their means fluctuate according to Figure 4.3. Once requests are generated, they are then immediately forwarded to the corresponding and available service load balancers.

We compare the proposed policy against the following policies, which are oblivious to the variability of the workload, heterogeneity of VM throughput, or billing periods:

- Policy I: statically providing maximum number of VMs for each service. This is a workload oblivious policy.

- Policy II(a), Policy II(p): dynamically providing VMs for each service, according to the workload only. This is a purely workload-driven policy. The number of VMs is decided by Equation 4.5, but based on the minimum VM throughput only. We provide two versions of Policy II, namely II(a), and II(p). The former one uses the actual request rate information and the later one uses the predicted ones in Equation4.2.

We also use the actual and predicted invocation rates in the proposed broker, and denote them broker(a) and broker(p) in the following. The difference be-

tween these two versions comes from the performance degradation due to the inaccurate workload prediction.

4.3.3 Two Services

We first evaluate the VM broker on a system with two services, namely service 1 and 2 (S_1, S_2), shown in Figure 4.3. We summarize the results in terms of cost saving, average utilization, average response time, and average normalized VM throughput in Table 4.1. The cost savings are compared with the cost of policy I, where the provisioning costs are the highest. The normalized VM throughput is calculated from the observed VM throughput divided by the minimum throughput for each service.

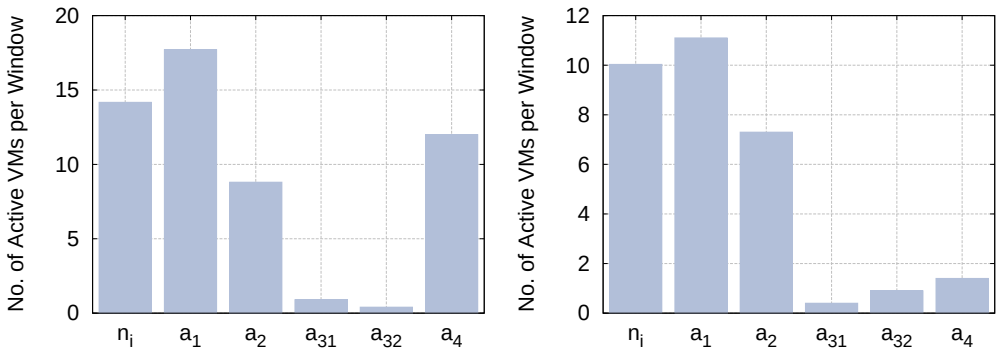
Clearly, static provisioning of VMs in policy I incurs high costs, and results in low utilization of VMs, as well as lower response times. Policy II saves costs for both services, compared to the policy I, because of frequently turning VMs on and off, and being oblivious to the performance variability of VM throughput. However, policy II has a much lower cost saving than the broker, especially for service 1, whose workloads are more stable and the cost savings from dynamic VM provisioning is smaller. In contrast, the broker can leverage the control knob of "replacing" VM for less varying workloads and acquire faster VMs, which in turn results in a lower number of VMs provisioned and subsequently lower cost. As for the utilization, the broker maintains the utilization at roughly 70 %, which is slightly lower than the target value of 80 %. Overall, the broker has the highest cost savings, medium utilization, the lowest response time, and the highest normalized VM throughput.

Furthermore, we present average statistics about control actions and the number of VMs provisioned in Figure 4.4. We denote a1, a2, a31, a32, and a4 as the number of VM, which are turned off, replaced, and reconfigured to another services, reconfigured from another service VM, and turned on, respectively. The first three actions are associated with "decreasing VMs", while the latter two are for "increasing" VMs. One can see that there is a higher number of reconfiguration and lower number of replacing occurring for service 2, because of a higher oscillation in workloads compared to service 1. Due to a higher variety of control actions taken for service 2, the broker is able to obtain VMs with higher throughput, i.e., the average normalized throughput of service 2 is higher than service 1. We conclude that the broker can apply different control actions according to different dynamics of workloads, and further gain cost savings without sacrificing the performance.

Table 4.1. Performance of different policies: two services.

Policy	Total	S_1				S_2			
	CS[%]	CS[%]	U[%]	RT[s]	AT	CS[%]	U[%]	RT[s]	AT
I	0	0	55.8	0.91	1.14	0	49.4	0.45	1.15
II(a)	14	7	74.6	1.08	1.16	21	74.9	0.55	1.17
II(p)	14	7	74.1	2.18	1.17	21	73.7	0.55	1.16
Brok.(a)	32	27	68.9	0.96	1.22	38	72.7	0.52	1.23
Brok.(p)	31	25	68.0	1.75	1.22	37	71.6	0.50	1.23

cs=normalized cost savings, U=utilization,
RT=average response time of invocation, AT=average normalized throughput of VMs



(a) Service 1 (b) Service 2

Figure 4.4. Average number of active VMs per window(n_i) and average number of VM used in each control action: turn off (a1), replace (a2), reconfigure off (a31), reconfigure on (a32), turn on (a4).

4.3.4 Four Services

Secondly, we evaluate the VM broker on a system with four services, namely service 1-4 (S_1, S_2, S_3, S_4), shown in Figure 4.3. We present the cost savings, average utilization, average response time, and average normalized VM throughput under different policies in Tables 4.2 and 4.3. The statistics of the control actions used in the proposed broker are summarized in Figure 4.5. Similar to the observations made in previous sections, we can see that the proposed VM broker can achieve significantly higher costing savings than other policies, while adhering to the utilization target and attaining lower response times. For the services with less varying workloads, i.e., services 1, 3 and 4, the broker can gain cost saving by replacing slower VM with faster VMs, whereas policy II can only gain marginal cost savings, compared to the static provisioning. For service 2, although policy II and the broker can gain good cost savings, the broker still obtains twice as high

Table 4.2. Performance of under different policies: four services, services 1 and 2

Policy	Total	S_1				S_2			
	CS[%]	CS[%]	U[%]	RT	AS	CS[%]	U[%]	RT	AS
I	0.00	0.00	55.8	0.91	1.14	0.00	49.4	0.45	1.15
II(a)	12.4	7.95	74.6	1.09	1.15	20.2	74.9	0.56	1.15
II(p)	11.2	5.61	73.9	2.37	1.16	19.7	73.9	0.56	1.15
Brok.(a)	29.7	30.1	71.6	0.99	1.22	38.8	73.2	0.52	1.24
Brok.(p)	29.5	28.4	71.6	2.34	1.20	37.0	71.8	0.51	1.23

CS=normalized cost saving, U=utilization, RT=average response time of invocation
AT=average normalized throughput of VMs

Table 4.3. Performance of under different policies: four services, services 3 and 4

Policy	Total	S_3				S_4			
	CS[%]	CS[%]	U[%]	RT	AS	CS[%]	U[%]	RT	AS
I	0.00	0.00	63.8	0.37	1.16	0.00	58.2	1.74	1.17
II(a)	12.4	7.18	75.0	0.44	1.16	13.3	76.0	1.98	1.17
II(p)	11.2	7.01	75.1	0.44	1.18	12.0	76.7	2.17	1.16
Brok.(a)	29.7	19.6	73.1	0.42	1.23	28.6	74.9	1.86	1.23
Brok.(p)	29.5	21.9	74.0	0.42	1.25	29.3	75.6	2.03	1.23

CS=normalized cost saving, U=utilization, RT=average response time of invocation
AT=average normalized throughput of VMs

savings as policy II due to effective replacing and reconfiguration of VMs.

As the broker applies VM control actions in a collaborative manner, especially through the reconfiguration, a better performance gain can be achieved by the broker in systems with a higher number of services. The overall performance in the case of four services is better than in the case of two services. One can observe this especially by comparing service 1 and 2 in both system scenarios. In particular, the average utilization of VMs for each service increases slightly and gets closer to the target value (80%). Both services also achieve higher cost savings in the four services scenario, because of more opportunities to reconfigure VMs to different services. From our evaluation, we believe our proposed policy can opportunistically acquire a fewer and faster VMs for different workloads, and its effectiveness grows with the scale of the system, i.e., with a higher number of different services.

Discussion: We would like to point out a few limitation of our study. First, this study considers only atomic services, and modeling dependencies among services (i.e., composite services) will be our future work. Second, we adopt preset values for modeling the variability of VM performance. We note that the cost savings and performance metrics presented here can change depending on those values. We plan to conduct extensive measurements in a commercial cloud

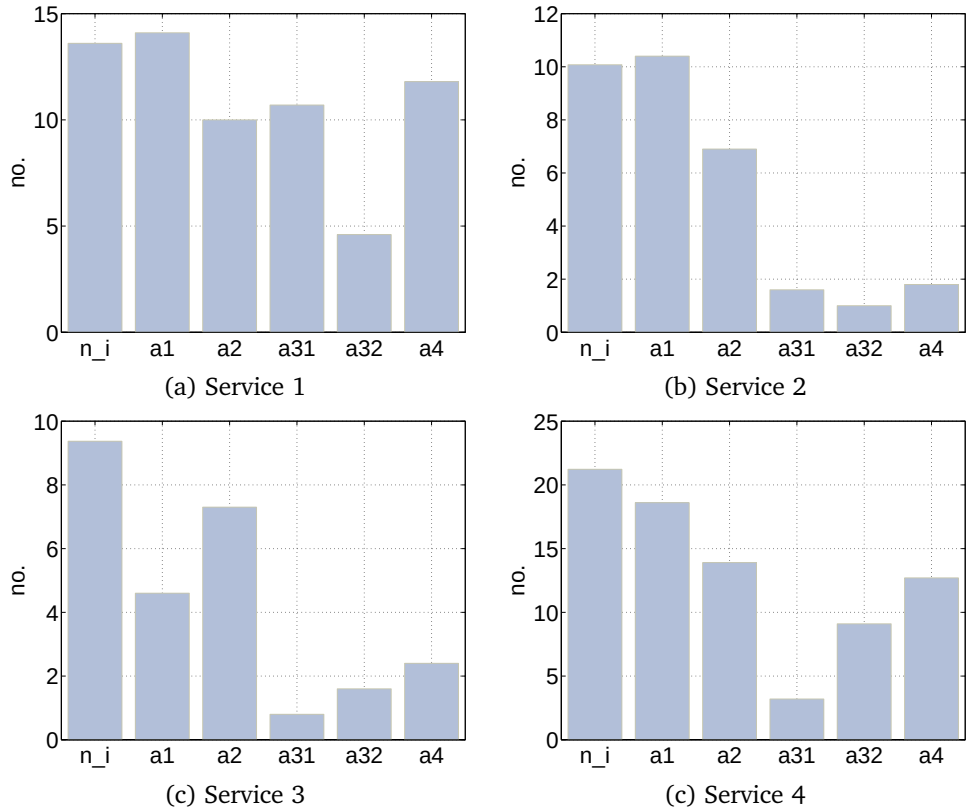


Figure 4.5. Average number of active VMs per window(n_i) and average number of VM used in each control action: turn off (a1), replace (a2), reconfigure off (a31), reconfigure on (a32), turn on (a4).

environment to confirm that our simulation results can be carried over to real systems.

4.4 Assumptions and Limitations

As described in Section 4.1, we only consider atomic, stateless services in this work. Furthermore, the assumption of CPU-bound services implies that the services can easily be reconfigured and migrated, without e.g., heavy dependence on data that would also need to be transferred in order to start up a new service VM instance. Taking into account more complex services with different resource requirements would complicate the opportunistic replication algorithm. More complex services would also potentially limit the degree to which the performance variability could be exploited, due to more overhead for reconfiguring

and replacing VMs.

4.5 Summary

In this chapter, we proposed an opportunistic replication policy especially designed for services deployed in a cloud. The objective of our work was to leverage the variability in VM performance and pay-as-you-go billing contracts in the cloud, such that the number of VMs for each service is minimized and opportunistically provisioned with better performing VMs. Our policy is based on comprehensive workload and system characteristics, i.e., time variability of workloads, VM variability, invocation variability, and billing periods. Moreover, we considered a complex set of control actions, i.e., turning VMs on and off, replacing VMs, and reconfiguring VMs, with detailed modeling of the respective overhead. The proposed policy not only optimizes the cost of a single service but also the welfare of all services in a collaborative manner.

Our evaluation results using production traces showed that the proposed policy achieves lower cost and better performance in terms of VM utilization and response time, compared to existing replication policies that are oblivious to performance and billing characteristics of the cloud.

Chapter 5

Providing Tail Throughput QoS Guarantees

Recent studies show that service systems hosted in clouds can elastically scale the provisioning of pre-configured virtual machines (VMs) with workload demands, but suffer from performance variability, particularly from varying response times. Service management in clouds is further complicated especially when aiming at striking an optimal trade-off between cost (i.e., proportional to the number and types of VM instances) and the fulfillment of quality-of-service (QoS) properties (e.g., a system should serve at least 30 requests per second for more than 90% of the time). Several empirical studies [Xu et al., 2013; Schad et al., 2010; Casale and Tribastone, 2013] point out a common pitfall in clouds that the performance variability — in this case the response time of services — fluctuates significantly, and tail latency degrades.

In this chapter, we develop a QoS-aware VM provisioning policy for service systems in clouds with high capacity variability, using experimental as well as modeling approaches. Using a wiki service hosted in a private cloud, we empirically quantify the QoS variability of a single VM with different configurations in terms of capacity. We develop a Markovian framework which explicitly models the capacity variability of a service cluster and derives a probability distribution of QoS fulfillment. To achieve the guaranteed QoS at minimal cost, we construct theoretical and numerical cost analyses, which facilitate the search for an optimal size of a given VM configuration, and additionally support the comparison between VM configurations.

This study aims to find the optimal VM provisioning for a service system, i.e., composed of an ideal VM configuration using a minimum number of VM instances, such that the required QoS properties are guaranteed for a certain frac-

tion of time at minimal cost (e.g., 90% of the time the sustainable throughput should be at least 30 requests per second). To such an end, we study a Wikipedia service [Wikipedia, 2014] and first empirically quantify its capacity variability on different VM configurations, in the presence a daemon VM executing various benchmark workloads in a private cloud. Leveraging our empirical experience, we build a Markovian model which explicitly models the capacity variability of an entire cluster, and we derive the *probability distribution* of the delivered QoS for a given number of VMs of a certain configuration. Based on analytical solutions regarding the QoS fulfillment, we construct theoretical and numerical analyses to evaluate the tradeoff between cost and the fulfillment of QoS promises, (1) by comparing optimal provisioning to simple pessimistic and optimistic provisioning; (2) when provisioning based on the average capacity fails; and (3) when choosing a VM configuration that returns the best cost/service-availability ratio.

Our contributions are two-fold: Firstly, we quantitatively characterize the capacity variability of a VM running a wiki service against a co-located VM running various workloads in a controlled private cloud environment. Secondly, we build a Markov-chain model to answer the question of how to guarantee ξ% of QoS fulfillment, i.e., avoiding tail performance degradation. Based on our experiments and model, we can dimension a cluster of VMs and choose among different VM configurations, such that the best trade-off between cost and QoS fulfillment is achieved.

In this work, we analyze a cluster of VMs running a *single service type*, i.e., a wiki service. The wiki service is assumed to be *stateless*, and additional VMs running the same wiki service can be started or stopped at any time to dimension the cluster according to the observed and predicted demand.

This chapter is organized as follows: The capacity variability of a VM hosting a wiki service on different VM configurations is discussed in Section 5.1. The proposed Markovian model and VM provisioning optimization is described in Section 5.2. Section 5.3 presents our cost analysis, and Section 5.4 lists the assumptions and limitations of this work. Section 5.5 summarizes this chapter.

5.1 Capacity Variability of Service VM Configuration

In this section, we use a controlled cloud environment to study the capacity variability of service hosting on different VM configurations, i.e., the fluctuation of capacity, against single neighboring VMs executing various workloads. To such an end, our target service is a Wikipedia deployed on a set of VM configurations and co-located with a daemon VM executing Dacapo benchmarks [DaCapo,

2014] in a private cloud. Essentially, we use the daemon VM to synthesize interference that can be encountered by a wiki VM in the cloud and parameterize the capacity variability, which is then used to build the QoS model for a service cluster in Section 5.2.

5.1.1 Experiment Setup

From our private cloud environment, we chose two IBM System x3650 M4 machines, *gschwend* and *nussli*, each with 12 Intel Xeon E5-2620 cores running at 2.00GHz, and 64 and 36 GB of RAM, respectively, for running our experiments. We use KVM on gschwend for hosting our target and daemon VMs, and nussli for generating the Apache JMeter workload requests for our target wiki VM.

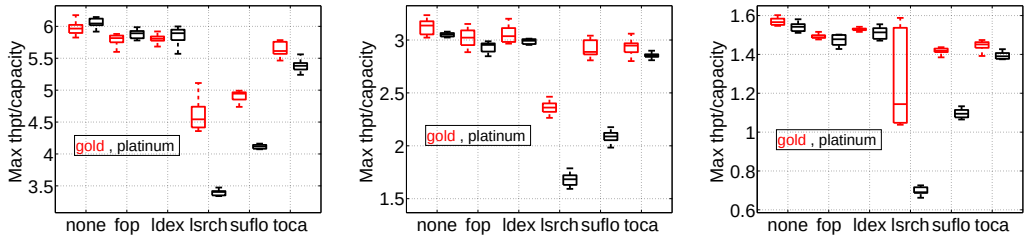
The target Wikipedia system is based on a subset of 500000 entries from a *pages-articles.xml* dump downloaded on October 12, 2012. The wiki VM is a Debian 7.0 system running an Apache 2.4.4 web server, the PHP 5.4.15 server-side script engine, MediaWiki 1.21 as the web application, and the MySQL 5.5.31 database. The number of threads employed by Jmeter is configured such that the maximum throughput of the wiki VM is reached. As for the workload on the daemon VM, we selected the following benchmarks from the Dacapo benchmark suite: (1) `fop`, a lowly threaded CPU-intensive benchmark; (2) `luindex`, a lowly threaded IO-intensive benchmark; (3) `sunflow`, a highly threaded CPU-intensive benchmark; (4) `lusearch`, a highly threaded CPU- and IO-intensive benchmark; and (5) `tomcat`, a network-intensive benchmark. We refer readers to [Chen et al., 2012] for the detailed threading behaviors and characterization of the Dacapo benchmarks.

We consider four types of VM configurations, with CPUs and memory sizes as listed in Table 5.1, which are comparable to VM offerings in Amazon EC2 [EC2, 2014]. We use three configurations for the wiki VM (bronze, silver, and gold), and two configurations for the daemon VM (gold and platinum). Based on experimental evaluation, we use two, four, and eight threads when running Jmeter against a wiki running on a bronze, silver, and gold VM instance, respectively. In total, we evaluate the amount of performance interference experienced by the wiki under 36 scenarios, i.e., three configurations of wiki VMs × six types of daemon workloads (5 DaCapo benchmarks and no workload) × two daemon VM configurations.

The target wiki performance statistics are collected from the Apache log files which record the current time, the requested URL, and response time for each request. After a warm-up period for the wiki VM, Jmeter, the daemon VM and the DaCapo benchmark, we start collecting statistics for five minutes for each of the

Table 5.1. VM configurations and naming conventions

	Bronze	Silver	Gold	Platinum
No. processing units	1	2	4	8
RAM (GB)	4	8	16	32



(a) Wiki on gold (b) Wiki on silver (c) Wiki on bronze

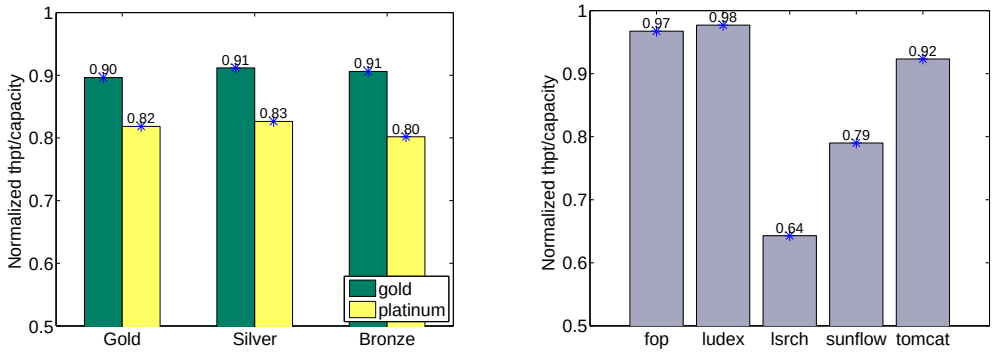
Figure 5.1. Capacity variability of a wiki running on different VM configurations against `fop`, `luindex`, `lusearch`, `sunflow`, and `tomcat`, hosted on gold and platinum VMs: box plots based on 10 repetitions.

36 scenarios, each of which is repeated ten times. We summarize the results of $36 * 10 = 360$ runs using box plots in Figure 5.1. One can straightforwardly find that the capacity variability of the wiki, i.e., the difference between no workload and different DaCapo benchmarks running on the daemon VM, can vary significantly depending on VM configurations and the characteristics of the DaCapo benchmark.

For further analysis, we take the median of the repeated runs of all scenarios and compute the average of the normalized throughput, compared to the scenario with no daemon VM neighbor. We thereafter categorize the results by target VM type, daemon VM type, and benchmark.

5.1.2 (In)sensitivity of Capacity Variability

To compare the robustness of different target VM configurations, we normalize the throughput of the wiki VM by the throughput of the wiki without any neighbor for gold, silver, and bronze VMs. In Figure 5.2(a), we present the average normalized throughput, a higher value of which means less interference is observed and the wiki VM is more robust. When co-located with a gold daemon VM, the difference between wikis running on different VM configurations is almost negligible. However, in our setup, when the daemon VM is more dominant, i.e., a platinum VM, a wiki on a silver VM seems to be slightly more robust than



(a) Impact of different target VM configurations (b) Impact of different daemon workloads

Figure 5.2. Average analysis of normalized throughput of target wiki.

when on a gold or bronze VM. Such an observation can also be made for individual daemon workloads, see Figure 5.1. Overall, our experiments show that a wiki running on a silver VM is slightly more robust to noisy neighbors, and the capacity of the wiki can be throttled by 10-20% on average due to interference from neighboring VMs.

5.1.3 A Really Noisy Daemon

We try to identify which type of workload represents the noisiest neighbor and causes high capacity variability for a wiki service co-located on the same physical machine. We compute the average normalized throughput across all target VM configurations for each benchmark, as presented in Figure 5.2(b). One can clearly see three levels of performance variability: (1) mild interference from `fop`, `luindex`, and `tomcat`, where the capacity degradation is within 10%; (2) medium interference from `sunflow`, where the capacity is degraded by roughly 20%; and (3) high interference from `lusearch`, where the capacity degradation can be up to 35%. Clearly, `lusearch` is the noisiest neighboring VM for our wiki service, as they both compete for a similar set of resources, i.e., both CPU and IO. As both `fop` and `luindex` have limited concurrent threading, only limited performance interference is observed.

Up to this point, our experiments have addressed the variability of a wiki service hosted on a single VM. In the next section, we leverage Markovian modeling to capture the capacity variability of a wiki cluster consisting of multiple VMs.

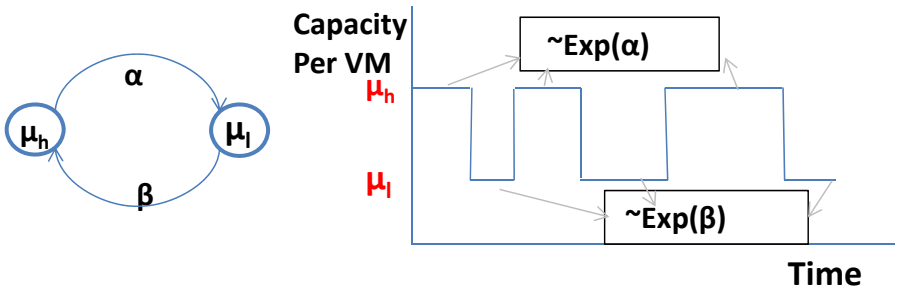


Figure 5.3. Capacity variability of a VM: state diagram of high and low capacity (left) and illustration of time series (right)

5.2 Markov Chain Model for Service Cluster

In this section, our objective is to derive a rigorous mathematical analysis for answering the question, "what is the minimum size of a cluster whose VMs experience capacity variability such that the probability of achieving a target QoS is guaranteed?". We define the service capacity $C(n)$ as the total number of requests processed by a cluster of $n \in \mathbb{Z}$ VMs, its QoS target as C^*, the fulfillment of which should be above a certain threshold ξ. Using Markov chain modeling, we obtain the steady-state distribution of QoS of a cluster with n VMs, and further search for the minimum n that satisfies the desired availability, $Pr[C(n) > C^*] > \xi$.

We start out the analysis by modeling the transition between high and low capacity of a single wiki VM, using values obtained in the previous section. Based on that, we develop a continuous-time Markov chain to model the service availability of the entire cluster. Finally, we show, by theoretical analysis and numerical examples, that the proposed minimum cluster size, n^*, indeed attains a good trade-off between cost and guarantee of service availability.

5.2.1 Single VM node

We assume that a VM of a certain configuration (e.g., gold, silver, or bronze) alternates between states of high and low capacity, denoted by μ_h and μ_l, for exponentially distributed times with rate α and β, respectively. Examples of such values can be found in Figure 5.1 for different VM configurations. We term the difference between μ_h and μ_l the capacity variability, and (α, β) the intensity of the variability. Figure 5.3 illustrates the state transitions and time series of such a model. To capture the maximum variability possibly experienced by a

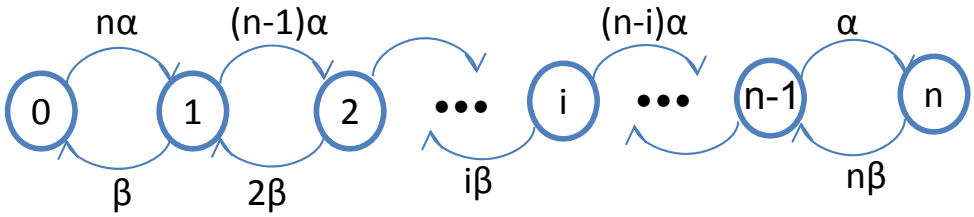


Figure 5.4. Markov chain of aggregate VM capacities, where the state denotes the number of VMs experiencing low capacity.

VM, we only adopt two states of capacity, namely high and low, for different VM configurations. Their parameterizations can be carried out by our empirical analysis in Section 5.1. On the contrary, the values of α and β depend on the workload dynamics of the underlying cloud, and thus are assumed invariant to VM configurations. Note that one may find intermediate states in reality, i.e., the capacity is between $[\mu_l, \mu_h]$. Our proposed model can be further refined to accommodate multiple levels of capacities, albeit with a higher computation overhead for obtaining steady-state probability of service availability (see the next subsection).

5.2.2 Continuous Markov Chain Modeling of the Cluster

The single VM model naturally leads us to use a continuous-time Markov chain (CTMC) to describe the dynamics of available capacity in a cluster consisting of n VMs, experiencing high and low capacity. In the proposed CTMC, a state $i \in I = \{1, 2, \ldots n\}$ is defined as the number of VMs having low capacity, while the rest of $n-i$ VMs in the cluster have high capacity. Consequently, the corresponding capacity of state i in the systems is

$$C_i(n) = i\mu_l + (n-i)\mu_h.$$

Note that $C_i(n) \geq C_j(n)$, for $i \leq j$ — essentially, $C_i(n)$ monotonically decreases in i. When there are i VMs with low capacity, the system transpositions to state $i+1$ with the rate $(n-i)\alpha$, and to state $i-1$ with the rate $i\beta$. Figure 5.4 illustrates such a Markov chain for a cluster of n VMs.

We let $\pi = [\pi_0, \pi_1 \dots \pi_n]$ denote the steady-state probability that the system has a service capacity of $C_i(n)$. One can solve the Markov chain in Figure 5.4 by a set of balance equations [Nelson, 1995], i.e.,

$$(n-i)\alpha\pi_i = (i)\beta\,\pi_{i+1}\ \forall i,$$

$$\sum_i \pi_i = 1.$$

Substituting all π_i as a function of π_n, we can then obtain the closed formed solution of π

$$\begin{aligned} \pi_n &= \frac{1}{(1+\frac{\alpha}{\beta})^n} \\ \pi_i &= \binom{n}{i}(\frac{\alpha}{\beta})^{n-i}\pi_n,\ 0 \le i < n. \end{aligned} \tag{5.1}$$

Consequently, we can derive the probability that the service capacity is greater than the target

$$\mathbf{Pr}[C(n) > C^*] = \sum_{i \in \{I : C_i(n) > C^*, i \le n\}} \pi_i. \tag{5.2}$$

To compute $\mathbf{Pr}[C(n) > C^*]$ for all $n \in \mathbb{Z}$, one shall first compute the values π_i, $\forall i$ using 5.1 for a given n, and the sum of π_i for the states i where the resulting capacity is greater than C^*, and then iterate the computation procedure for all values of n.

5.2.3 Trade-off between Cost and Service Availability

To find a minimum cluster size that ensures that a service capacity greater than the target capacity, $C > C^*$, is guaranteed for ξ% of time, we can formulate the following optimization after substituting Equation 5.1 into the constraints and rearrangements:

$$\begin{aligned} \text{minimize} \quad & n \\ \text{subject to} \quad & (n-i)\mu_h + i\mu_l \ge C^* \\ & \sum_i \binom{n}{i}(\frac{\alpha}{\beta})^{n-i}\frac{1}{(1+\frac{\alpha}{\beta})^n} \ge \xi \\ & i \le n \end{aligned}$$

For given values of α, β, μ_h, and μ_l, $Pr[C(n) > C^*]$ is a function increasing in n, i.e., when $n_1 \ge n_2$, $\mathbf{Pr}[C(n_1) > C^*] \ge \mathbf{Pr}[C(n_2) > C^*]$, as self-explained in

the second constraint in the above optimization. Consequently, one can straightforwardly find the optimal n^* by linearly searching through the possible values of $n \in \mathbb{Z}$ in an increasing order.

Note that the optimization is constructed implicitly depending on the workload intensity via the value of C^*. For a given period of time when the workload intensity is predicted as λ requests per second, one may want to keep the system 80% utilized, and set the target capacity to $C^* = \lambda/0.8$. For more discussion on the choice of the target capacity, see Chapters 3 and 4.

n^* vs. Simple Solutions

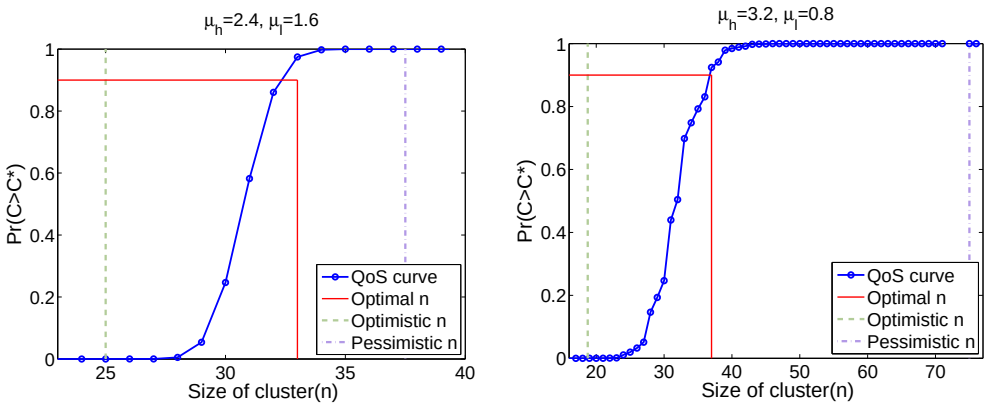
Herein, we illustrate how n^* obtained through our proposed methodology attains a good trade-off between the cost and the guaranteed service availability, compared to simple optimistic and pessimistic solutions. One may optimistically think that all VMs have high capacity and only purchase $n^{opm} = \lceil C^*/\mu_h \rceil$ VMs by simply dividing the target capacity with the value of high capacity of a single VM. In contrast, a pessimistic solution would be to assume that all VMs have low capacity and purchase $n^{psm} = \lceil C^*/\mu_l \rceil$. As $\mu_h > \mu_l$, n^{psm} is greater than n^{opm}.

We compute the service availability curves by Equation 5.2 for all values of n that fulfill the target capacity of $C^* = 60$ requests per second, using $\alpha = 60$, $\beta = 50$ and two sets of μ_h and μ_l, respectively. Figure 5.5 summarizes the numerical results. Additionally, we also graphically illustrate the optimal provisioning of VMs (n^*) that fulfill the desired service availability, i.e., the cluster capacity is greater than 60 for $\xi = 90\%$ of the time, compared with pessimistic (n^{psm}) and optimistic (n^{opm}) solutions. We consider service availability curves in two cases of capacity variability, namely with smaller and bigger difference between the high and low capacity of a VM. One can easily see that the optimal cluster size grows with the variability, indicated by a higher value of n^* in Figure 5.5(b) than (a). When the variability of capacity is higher, the service availability curve increases slower in n than in the low variability case. Moreover, the pessimistic and optimistic allocations are even further away from the optimal one.

To proceed to cost comparison, we assume the cost of a cluster, $cost(n)$, is a strictly increasing function in n, i.e., $cost(n_1) \geq cost(n_2)$ when $n_1 \geq n_2$. Furthermore, due to the monotonicity of $Pr[C(n) > C^*]$ and $n^{opm} \leq n^* \leq n^{psm}$, we reach the following corollary:

Corollary 5.2.1.

$$\begin{aligned} &cost(n^{opm}) \leq cost(n^*) \leq cost(n^{psm}), \\ &\mathbf{Pr}[C(n^{opm}) > C^*] \leq \mathbf{Pr}[C(n^*) > C^*] \simeq \xi \leq \mathbf{Pr}[C(n^{psm}) > C^*]. \end{aligned} \tag{5.3}$$



(a) Smaller variability (b) Bigger Variability

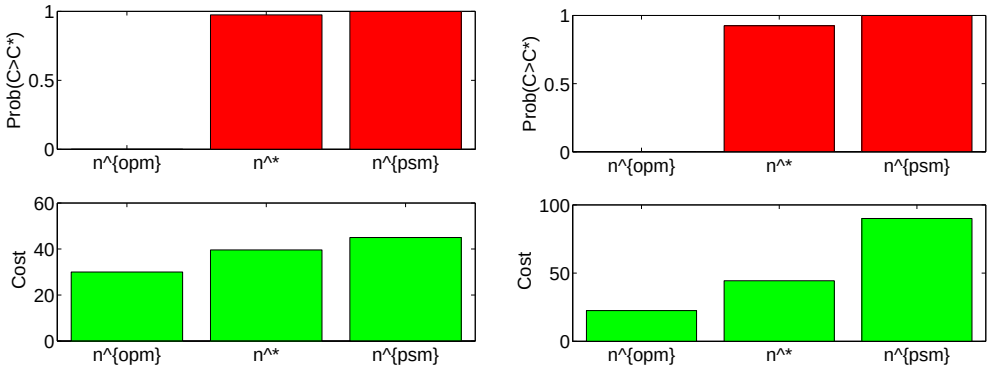
Figure 5.5. Service availability curve, $Pr[C(n) > 60]$: the optimal number of VMs to achieve $\xi = 90\%$, the pessimistic, and the optimistic solution.

Though the optimistic solution incurs lower cost, the QoS fulfillment threshold is not met. On the contrary, the pessimistic solution can achieve the service availability with 100% guarantee, but at a higher cost. The optimal provisioning of VMs, n^*, indeed achieves a good trade-off between cost and QoS fulfillment, compared to simple optimistic and pessimistic solutions. Note that n^* can result in a slightly higher value of $\mathbf{Pr}[C(n^*) > C^*]$ than ξ, due to the discrete choice of the number of VMs.

We further numerically illustrate how such a trade-off is affected by different levels of variability in capacity of a single VM. Using a simple linear cost function, i.e., $cost(n) = 1.2 \cdot n$, we construct two numerical examples in Figure 5.6, following the parameters discussed in Figure 5.5. Note that the cost here is defined as the cost per time unit, which can be aligned with the billing periods used in commercial clouds, e.g., one hour. One can see that n^* can improve the QoS fulfillment drastically by increasing cost, compared to n^{opm}, and reduce cost significantly by allowing a fractional capacity degradation, compared to n^{psm}. The advantage of n^* in attaining a good trade-off is even more prominent in the case of bigger variability.

Why not Consider Average Capacity of a VM?

In this subsection, we show that choosing n based on the average capacity of a VM cannot reach the optimal values nor guarantee QoS fulfillment at the target capacity level, using numerical examples. Recalling the state transition of a VM depicted in Figure 5.3(a), the average capacity of a single VM, μ, and the VM



(a) Lower variability, $(\mu_l, \mu_h) = (1.6, 2.4)$

(b) Higher variability, $(\mu_l, \mu_h) = (0.8, 3.2)$

Figure 5.6. QoS fulfillment vs. cost: $\mathbf{Pr}[C(n) > C* = 60] > \xi = 0.9$

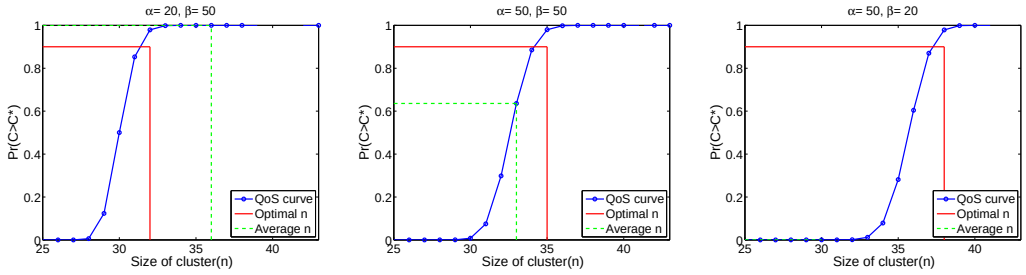
provisioning based on the average capacity, n^{avg} are

$$\mu = \frac{\mu_h \alpha + \mu_l \beta}{\alpha + \beta}, \text{and } n^{avg} = \frac{C^*}{\mu},$$

respectively. Figure 5.7 demonstrates that a cluster size based on the average capacity is not a reliable solution under three scenarios of (α, β), namely (a) often experiencing low capacity (b) alternating between high and low capacity equally, and (c) often experiencing high capacity. We let $\mu_h = 2.4$ and $\mu_l = 1.6$, as used in the case of small variability. Shown in Figure 5.7(a), when $\alpha < \beta$, n^{avg} tends to overestimate and $\mathbf{Pr}[C(n) > C^*]$ is over the required values, $\xi = 0.9$. When $\alpha > \beta$, n^{avg} tends to underestimate and $\mathbf{Pr}[C(n) > C^*]$ is below the required values, indicated by the horizontal line overlapped on the x-axis in Figure 5.7(c).

As for $\alpha = \beta$, we want to highlight that n^{avg} can achieve the target capacity roughly 50% of the time, for any capacity variability and target values. This observation can be explained by Equation 5.1. When $\alpha = \beta$, the steady state of QoS fulfillment is greatly simplified to $\pi_n = 1/2^n$ and $\pi_i = \binom{n}{i}(1/2^n)$. Thus, substituting $n^{avg} = \lceil 1/2\mu_h + \mu_l \rceil$ can result in $\mathbf{Pr}[C(n) > C^*] = (50+\epsilon)\%]$, where ϵ is a small positive fluctuation due to the ceiling operator on n^{avg}.

Observation 5.2.2. *When $\alpha = \beta$, n^{avg} can achieve $C(n) > C^*$ roughly 50% of the time, i.e., $Pr[C(n) > C^*] = 50 + \epsilon\%$, where ϵ is a small positive value.*



(a) $\alpha < \beta$ (b) $\alpha = \beta$ (c) $\alpha > \beta$

Figure 5.7. QoS fulfillment curves based on n^{avg} and n^*, under $\mu_h = 2.4$ and $\mu_l = 1.6$.

5.3 Choosing a VM Configuration

In this section, we compare different VM configurations in terms of their optimal cluster sizes and total cost, based on our proposed Markov chain model. Using theoretical and numerical analysis, we study if a cluster composed of more powerful VMs is always smaller than a cluster of weaker VMs. Due to the large number of parameters considered, we focus on providing a condition where weaker VMs imply a bigger cluster, and numerical counter examples where a cluster of weaker VMs can provide better service availability than a cluster of more powerful VMs.

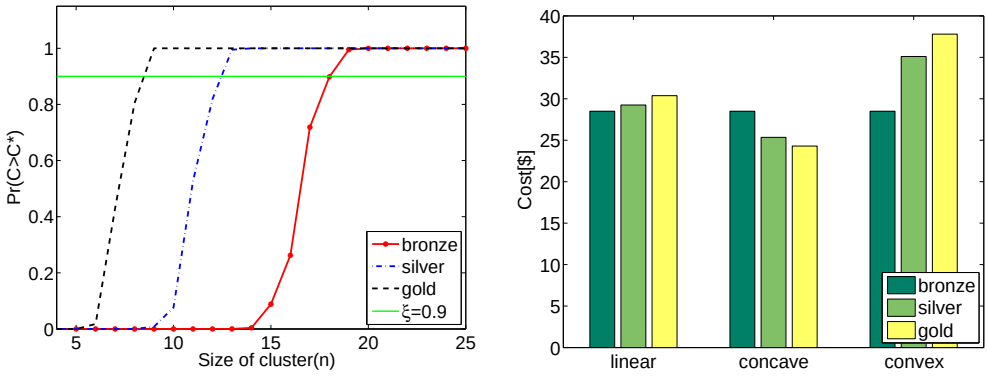
5.3.1 Typical Case: Weaker VM Means a Bigger Cluster

Following the convention in Section 5.1, we consider three types of VM instances, namely gold, silver, and bronze. A gold instance is more powerful and implies a higher average computational capacity than a silver instance, whose average capacity is more than that of a bronze instance. All VM configurations experience high ($\mu_{h,type}$) and low capacity ($\mu_{l,type}$) for exponentially distributed durations with means equal to α and β, respectively. We can show the necessary condition for the *typical case*, meaning clusters of weaker VMs are bigger than clusters of more powerful VMs when achieving the same target of service availability.

Theorem 5.3.1. *When experiencing the same α and β and aiming at the same service availability threshold, the cluster sizes of gold, silver, and bronze instances are*

$$n^*_{gold} \leq n^*_{silver} \leq n^*_{bronze}, when$$

$$\mu_{h,gold} \leq \mu_{h,silver} \leq \mu_{h,bronze}, and\ \mu_{l,gold} \leq \mu_{l,silver} \leq \mu_{h,bronze}.$$



(a) QoS Curves (b) Different cost functions

Figure 5.8. QoS fulfillment and cost comparison for a typical case: comparison of gold, silver and bronze VMs.

The theorem follows straightforwardly from the monotonicity of $Pr[C(n) > C^*]$ in n. Due to the lack of space, we skip the proof. The theorem tells us that to guarantee the same level of service availability, one should definitely acquire a higher number of weaker VMs than powerful VMs, when the low and high capacity of weaker VMs are inferior to the low and high capacity of powerful VMs, respectively.

We note that the typical case simply implies the order of n^* for different configurations, not the differences in their costs. Using three types of cost functions, namely linear, concave, and convex, we show that the costs of different types of VM clusters can vary a lot. In particular, the high and low capacities experienced by each VM configuration are listed under the typical case in Table 5.2, where (α, β) are (40,20). The linear/concave/convex cost function means the cost per VM instance is linearly/concavely/convexly proportional to the average capacity of single VM of a particular type. We set the cost per VM per time unit of (gold, silver, bronze) for linear, concave, and convex as (1.5, 2.25, 3.375), (1.5, 1.95, 2.7), and (1.5, 2.7, 4.2), respectively. Figure 5.8 (a) and (b) summarize the resulting service availability curves of different VM types and the resulting costs under different cost functions. One can see that although the bronze cluster is much bigger than the gold, the cost can still be lower when the cost per VM is linearly and convexly proportional to their average capacity. On the contrary, when there is a discount on computational capacity, i.e., when the cost per unit of computation decreases for gold, a gold cluster can be a cheaper option as shown by the case of a concave cost function.

Table 5.2. Capacity parameters of single VM for all VM types.

	Typical Case			Counter Example		
	μ_l	μ	μ_h	μ_l	μ	μ_h
Gold	3.75	4.5	5.62	0.26	2.65	5.03
Silver	2.25	3.00	3.75	0.95	2.30	3.64
Bronze	1.50	2.00	2.50	1.80	2.00	2.20

5.3.2 Counter Example: A Cluster of Weaker VMs Can Be Smaller

Here, we show by some counter examples that the optimal size of a cluster with weaker VMs is not necessarily larger. The capacity parameters of gold, silver, and bronze instances used are listed under the counter example in Table 5.2. The average capacity, μ, is the average of high and low capacity, and grows with the VM configuration. However, the capacity variability, i.e., the difference between high and low capacity, is higher for more powerful VMs.

Figure 5.9 summarizes the curve of QoS fulfillment of the three VM configurations. One can see that the QoS curve of the three types of VMs cross each other at $n = 15$. For a given size, the QoS of a gold VM is not necessarily higher than that of a silver or bronze VM. In particular, for $n \geq 15$, the QoS of a silver VM is higher or equal to a gold VM. As a result, depending on the threshold of QoS, ξ, the optimal cluster size of bronze VMs can be bigger, or smaller than that of gold VMs. To guarantee $Pr[C(n) > 30] \geq 0.85$, the optimal cluster size of all three types of VMs is 16. When such a threshold is higher than 0.85, the number of VMs in a gold cluster should be higher than in a bronze cluster. This leads us to conclude that not only the average, but also the variability in VM throughput is crucial in choosing and sizing VM clusters in the cloud.

Our proposed Markov model and solution provide an efficient means to explore a large number of parameters encountered, such as different cost functions, and exogenous variabilities and their intensity, when choosing the right VM configuration and deciding the cluster size. Numerical examples serve the purpose of illustrating how our solution robustly attains an optimal trade-off between cost and QoS fulfillment across different system parameters and VM configurations.

5.4 Assumptions and Limitations

In this work we assume a cluster of stateless VMs running a single service type, and in particular a wiki service. On the one hand, while this is a more realistic system than the ones described in Chapters 3 and 4 in the sense that the performance is not necessarily bound by a single resource type, we still assume

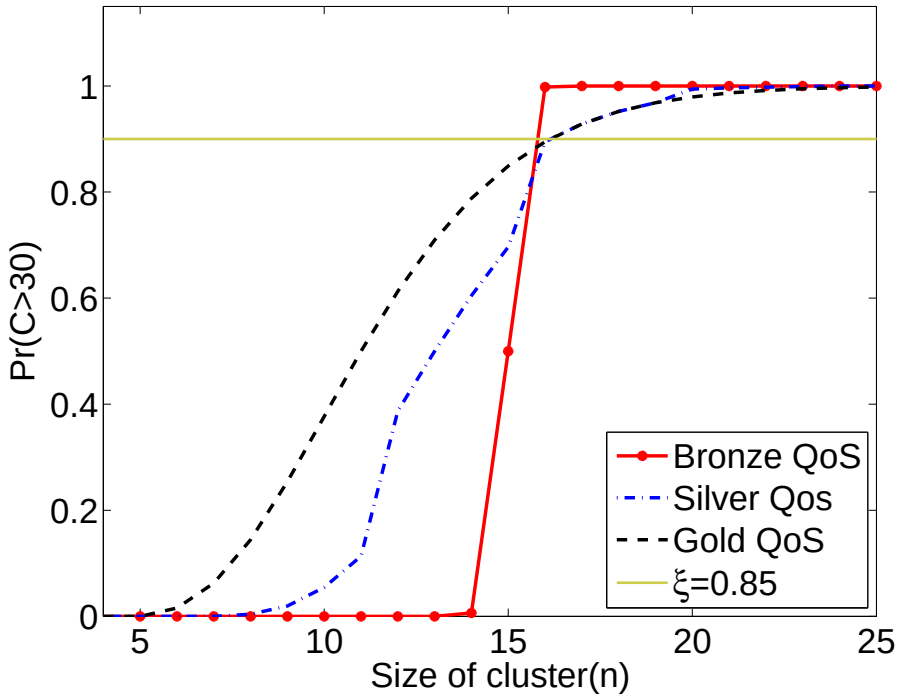


Figure 5.9. QoS curve of different types of VMs, under $\alpha = \beta = 20$, $Pr[C > 30] > 0.85$.

that new VMs can be started at any point of time. On the other hand, this work focuses more on a medium- to long-term capacity planning, where the overhead of transferring data necessary for starting new VM instances is less significant.

5.5 Summary

Using empirical experiments with a Wikipedia system, as well as a Markovian model and numerical analysis, we demonstrated how QoS fulfillment can be best guaranteed with a minimum number of correctly configured VMs deployed in a cloud where VMs suffer from high capacity variability. Our experimental results showed that different VM instance sizes can have varying degrees of capacity variability from co-located VMs and that workloads on co-located VMs can impact the capacity of the service VM by up to 35%. Our analytical and numerical results provided not only insight on how an optimal number of VMs should be chosen for a service cluster, but also give counter examples on why simple pessimistic, opti-

mistic, and average-based provisioning of VMs cannot strike an optimal balance of cost and QoS fulfillment in the cloud where performance variability persists. Overall, we provided a systematic and rigorous approach to explore several crucial aspects of VM provisioning for service clusters, i.e., capacity variability, cost structure, and guarantees regarding QoS fulfillment.

Chapter 6

Optimizing for Tail Response Times

Related work in the areas of queueing networks and operations research sheds light on analyzing either single or multiple aspects of deploying clusters in the cloud. Obtaining the distribution of response times is always challenging, especially with complex arrival and service processes, i.e., Markov modulated speed, and non- First Come First Serve (FCFS) scheduling. The only easy expression relates to the first moment of response times.

A large body of related work concentrates on deriving the distribution of the $M/G/1/PS$ queue [Kleinrock, 1975; Gautam, 2012], where the service rate is fixed. Considering varying service rates, the related work centers around two directions: (1) service rates depending on the state of the system [Rege and Sengupta, 1985; Gupta and Harchol-Balter, 2009] and (2) service rates changing due to external environmental processes [Mahabhashyam and Gautam, 2005; Boxma and Kurkova, 2001; Zhang and Zwart, 2012; Dorsman et al., 2013]. Motivated by the fact that the "speed" of a system increases with the number and variability of jobs, Gupta et al. [Gupta and Harchol-Balter, 2009] developed the approximation for mean response times for $M/G/PS-MPL$ and $GI/G/PS-MPL$ queues with state-dependent service rate, where MPL denotes the multi-programming limit. While most of the aforementioned work centers around single server/queue, Dorsman et al. [Dorsman et al., 2013] and Casale et al. [Casale and Tribastone, 2013] study parallel queueing networks with Markov-modulated execution speeds using a functional central limit theorem and ordinary differential equations, respectively, with a strong assumption on Markovian arrivals. Therefore, little is known on the challenging question of how to predict the tail response times for parallel queueing systems with renewal arrivals, high job size variability, Markov-modulated execution speeds, and processor sharing discipline under various traffic intensities.

Various strategies have been proposed to overcome the degradation of tail response times due to exogenous variability. On one hand, various opportunistic VM selection algorithms [Björkqvist et al., 2012; Farley et al., 2012] reactively obtain the VMs with better performance in terms of execution speed. On the other hand, cluster sizing algorithms aim to proactively provision VMs according to their predicted performance, such as the tail throughput [Björkqvist et al., 2013], and SLA targets. Though a pro-active optimization scheme is able to provide a reliable statistical guarantee on the performance metrics of interest, one needs to first obtain the prediction of those metrics, such as tail response times. Overall, optimizing for tail response times is achieved as best effort, without any guarantees.

In this chapter, we develop an abstract parallel queueing system, where each queue is a $G/G/1/PS$ with Markov modulated execution speeds, to represent the application cluster hosted in a cloud. We obtain the distribution of workloads accumulated in the system, with special focus on their tail, using large deviation analysis. To derive the approximation of the tail response times, we leverage the workload distribution and a mean-based analysis of the $M/G/1/PS$ queue with an average execution speed. We first derive the conditional distribution of the number of jobs for a given workload distribution for the $M/G/1/PS$ queue. Then we develop an approximation scheme for the tail response times – a kind of worst case analysis. As a special case to model highly varying job sizes, we present closed form results for a degenerate hyper-exponential distribution. We compare the proposed analysis against simulation results under various parameter settings, i.e., number of servers, different levels of exogenous variability (execution speeds), traffic intensities, and job size variability. Overall, our analysis shows a very good match with experimental results for the tail distribution of workloads, and response times, especially in cases with high job variability, number of speeds, and number of servers.

Our contribution can be summarized as follows. First, we derive the workload distribution for hard-to-analyze systems that capture the key characteristics of today's cloud systems, i.e., renewal arrivals, highly varying job sizes, Markov-modulated execution speeds, processor sharing, and round-robin load balancing. Second, we develop an approximation scheme for the tail response times, which are one of the critical SLA parameters, and further optimize the cluster size based on that.

The outline of this work is as follows. Section 6.1 provides an overview of the system model. In Section 6.2 we obtain the workload distribution based on the large deviation analysis for the $G/G/1/PS$ queue with Markov-modulated execution speeds and illustrate an approximation scheme to obtain the tail response

times. We develop a mean-based approximation in Section 6.3, which captures the conditional probability of the number of jobs and tail response times for the $M/G/1/PS$ queue with a special case on degenerated hyperexponential distribution of job sizes. Extensive experiments comparing analysis and simulation are given in Section 6.4, and Section 6.5 lists the main assumptions and limitations of our work. Finally, a summary is presented in Section 6.6.

6.1 System Model

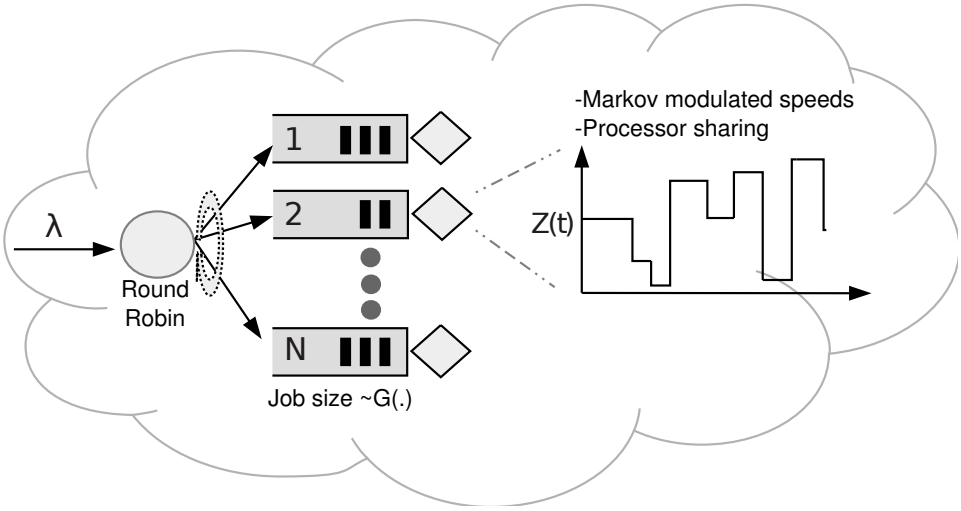


Figure 6.1. Cloud cluster scenario.

We consider a system of N parallel servers and a dispatcher in front of them, as depicted in Figure 6.1. Each server is functionally equivalent. Jobs arrive to the dispatcher according to a renewal process with a rate of λ per second, and an inter-arrival time squared coefficient of variation (SCOV) c_a^2. Each job requires a random amount of work (say, in KBytes) to be processed by one of the servers. We assume that the amount of work, denoted by H, for various arrivals is IID with a common CDF $G(\cdot)$, mean m and SCOV c^2. The dispatcher cannot observe the state of the servers, and hence routes jobs to the servers in a round robin fashion. The servers use complete processor sharing. However, the speed of the server changes according to a finite-state continuous time Markov chain (CTMC) $\{Z(t), t \geq 0\}$ with state-space $\mathscr{S}$ and infinitesimal generator matrix $\mathbf{Q}$. At any time t, if $Z(t) = i$, then the server uses execution speed ϕ_i, for any $i \in \mathscr{S}$. We

define the unit of ϕ_i as KBytes per second. If there are no jobs at the server at time t then the processor is idle, otherwise it serves jobs at speed ϕ_i. Note that the speed can change during any time of the job execution process, i.e., a job can be executed over multiple speeds. In summary, using Kendall's notation from queueing theory, each queue of our system is a $G/G/1/PS(\phi)$ queue, where $\boldsymbol{\phi}$ denotes the vector of all possible execution speeds.

This is a somewhat non-traditional description of a queueing system. Hence before proceeding ahead, we explain the service process in some detail. Say, at time t an arrival occurs to a server which already has $n-1$ jobs that it is processing in parallel. Let x_1, x_2, ..., x_n be the remaining amount of work (in KBytes) for jobs 1, 2, ..., n (where the n^{th} job corresponds to the one that just arrived at time t). Since the server's speed is ϕ_i, the amount of work for each of the n jobs would reduce at rate ϕ_i/n because of the processor sharing discipline. For instance, the k^{th} job has the smallest remaining work, i.e., $x_k = \min\{x_1, \ldots, x_n\}$. Then, the k^{th} job would be completed at time $t + nx_k/\phi_i$, provided that there are no arrivals as well as no state changes for execution speeds during the interval $(t, t+nx_k/\phi_i)$. If there are arrivals during the interval $(t, t+nx_k/\phi_i)$ but no state change, then as soon as the first arrival occurs, each job would now be processed at rate $\phi_i/(n+1)$. Conversely, if a state change but no arrival occurs during the interval $(t, t+nx_k/\phi_i)$, then immediately after the state changes (to say $j \in \mathscr{S}$) each of the n jobs would now get processed at rate ϕ_j/n.

Let $\mathbf{p}$ be the steady-state probability row-vector of the CTMC $\{Z(t), t \geq 0\}$ (assuming it is irreducible), i.e., $\mathbf{p}$ satisfies $\mathbf{pQ} = \overline{0}$ and $\mathbf{p}\overline{1}' = 1$ (where $\overline{0}$ and $\overline{1}$ are row-vectors of zeros and ones respectively). The condition for system stability is

$$\frac{\lambda m}{N} < \mathbf{p}\boldsymbol{\phi} \tag{6.1}$$

where $\boldsymbol{\phi}$ is a column vector of the ϕ_i values. Thus from inequality (6.1), we can immediately identify the minimum number of servers N that would result in a stable system.

Prior to formally introducing our problem, we first describe the definition for the so-called $(1-\epsilon)^{th}$ percentile of workloads and response times. For a random variable X, e.g., workloads and response times, the value $X_{1-\epsilon}$ is such that $P\{X \leq X_{1-\epsilon}\} = 1-\epsilon$. Thus $X_{1-\epsilon}$ is the $(1-\epsilon)^{th}$ percentile of X.

Problem statement: *What we look for is the minimum number of servers that would guarantee that the response time for an arbitrary arrival in steady state would be less that ζ with a probability greater than $1-\epsilon$, for some given values of ζ and ϵ, i.e., so-called $(1-\epsilon)^{th}$ response times.*

For that, we assume there are N servers that would result in a stable system, i.e., the inequality (6.1) is satisfied. Specifically, let R be the response time experienced by an arbitrary job that arrives in steady state to one of the N servers. We obtain an expression for $P\{R > \zeta\}$ and subsequently check if it is less than ϵ. If not, since our expression for $P\{R > \zeta\}$ is a function of N, it can be easily changed and we can find the smallest N so that $P\{R > \zeta\}$ is less than ϵ.

6.2 Tail Response Times of $G/G/1/PS(\boldsymbol{\phi})$ Queues

In this section we consider any one of the N servers, i.e., $G/G/1/PS(\boldsymbol{\phi})$ queues, and aim to obtain the (tail) response times. To such an end, we first derive the tail workload distribution based on large deviation analysis, and then derive approximation schemes of response times based on a mean-based approximation of the $M/G/1/PS$ queue with an average execution speed. It is crucial to point out that it is extremely difficult to obtain the response time distribution for the $M/G/1/PS$ queue. However, in this study we add some features beyond the $M/G/1/PS$ queue, such as Markov-modulated execution speed and round robin dispatching (resulting in general inter-arrival times). Undoubtedly, their analysis gets even more intractable, and hence we resort to an approximation scheme.

6.2.1 Workload Distribution

To obtain the tail distribution of the workload under round-robin load balancing, we use a method based on large deviations. Here we consider any one of the N servers. Arrivals occur at any server according to a delayed renewal process, with inter-renewal times according to a general distribution with mean N/λ and variance Nc_a^2/λ^2. Let W be the workload to be processed in the server at any arbitrary time in steady state. To obtain the limiting tail distribution of the workload in the queue, $W(t)$, i.e., $P\{W(t) > x\}$ as $t \to \infty$ for some large x, we consider a fictitious server with constant processing speed $\phi_{\max} = \max_{i \in \mathscr{S}} \phi_i$. In addition to the regular stream of arrivals, the fictitious server also gets fluid workload at rate $\phi_{\max} - \phi_j$ at time t (if $Z(t) = j$ for some $j \in \mathscr{S}$) from a compensating source. Note that the workload at time t at this fictitious server is stochastically identical to $W(t)$ for all $t \geq 0$. Thus we analyze the tail distribution of this fictitious server and represent it as $W(t)$.

Let $A(t)$ be the total amount of workload that arrived from time 0 to t from the regular renewal source. We can compute for some $\nu \geq 0$,

$$\begin{aligned} E[e^{\nu A(t)}] &= E[\{\tilde{G}(-\nu)\}^{\mathcal{N}(t)}] \\ &= e^{\frac{\lambda t}{N}\log\{\tilde{G}(-\nu)\}+\frac{\lambda t c_a^2}{2N^2}\{\log\{\tilde{G}(-\nu)\}\}^2} \end{aligned}$$

where $\mathcal{N}(t)$ is the number of arrivals in time t, assuming a non-delayed renewal process (this would not affect us as we will let $t \to \infty$), and for the latter term we use a Normal approximation to $\mathcal{N}(t)$ which is reasonable for large t. Now, we can write down $h(\nu)$ the asymptotic logarithmic moment generating function (ALMGF) defined as [Gautam, 2012]

$$h(\nu) = \lim_{t\to\infty} \frac{1}{t}\log E[e^{\nu A(t)}]$$

and for our $A(t)$, $h(\nu)$ can be computed as

$$h(\nu) = \frac{\lambda}{N}\log\{\tilde{G}(-\nu)\} + \frac{\lambda c_a^2}{2N^2}\{\log\{\tilde{G}(-\nu)\}\}^2 \tag{6.2}$$

for any $\nu \geq 0$. Likewise, since the compensating source is a CTMC fluid source, we can compute its ALMGF $h_c(\nu)$ as

$$h_c(\nu) = e(\mathbf{Q} + \nu\phi_{\max}\mathbf{I} - \nu\breve{}) \tag{6.3}$$

where $e(A)$ denotes the largest real eigenvalue of a square matrix A, $\mathbf{I}$ is the identity matrix, and $\breve{}$ is the diagonal rate matrix, i.e., $\breve{} = [diag(\phi)]$,

Let η be the unique solution to

$$h(\eta) + h_c(\eta) = \eta\phi_{\max}$$

where $h(\nu)$ and $h_c(\nu)$ can be computed from Equations (6.2) and (6.3), respectively. Then, using results from large deviations,

$$\lim_{t\to\infty} P\{W(t) > x\} \approx e^{-\eta x} \tag{6.4}$$

for large values of x (in particular as $x \to \infty$). Thus we have an approximation for the tail distribution of the steady-state workload.

6.2.2 Approximating Tail Response Times

Here, we seek to obtain an expression for the response time tail, based on the workload tail derived in Section 6.2.1. Notice that in Section 6.2.1 we do not

make any assumptions about the service discipline except that it is work conserving. Here, we focus on the tail probability of response time under a *processor sharing* discipline.

Assumptions: In particular, we assume fast dynamics of execution speeds, i.e., jobs can experience many different execution speeds during their response. The motivation behind is two-fold. First, co-located neighboring VMs can execute complex applications that exhibit volatile run-time behavior and cause frequent changes in execution speeds. Second, under a special case, it is possible to analytically derive the tail probabilities of response times under very fast dynamics: $\lambda/N \ll -\min_{j\in\mathscr{S}} q_{jj}$, where q_{jj} is the j^{th} element on the diagonal of $\mathbf{Q}$ matrix of CTMC. Essentially, the dynamics of the number of jobs in the system is much faster than the dynamics of execution speeds, i.e., a server changes state many times during a job's response. Especially for the scenario in [Reiss et al., 2012], this is a reasonable assumption.

Approximation Scheme

As described earlier, obtaining the tail response time for an $M/G/1/PS$ queueing system with fixed speed is itself not straightforward, and to the best of our knowledge not available in the open literature. Therefore, as a first step, we obtain an approximate expression for $R_{1-\epsilon}(M,\overline{\phi})$, the $(1-\epsilon)^{th}$ percentile of the response time under the $M/G/1/PS$ queue with an average speed $\overline{\phi}$ (where $\overline{\phi}=\mathbf{p}\boldsymbol{\phi}$). To obtain an expression for $R_{1-\epsilon}(M,\overline{\phi})$, we need a closed-form formula for $W_{1-\epsilon}(M,\overline{\phi})$, the $(1-\epsilon)^{th}$ percentile of the workload under the $M/G/1/PS$ queue with constant speed $\overline{\phi}$. However, what we ultimately need is an approximation for $R_{1-\epsilon}(G,\boldsymbol{\phi})$, the $(1-\epsilon)^{th}$ percentile of the response time under the $G/G/1/PS$ queue with varying speed $\boldsymbol{\phi}$. We already have an expression for the corresponding workload $W_{1-\epsilon}(G,\boldsymbol{\phi})$ from Equation (4), which can be written as

$$W_{1-\epsilon}(G,\boldsymbol{\phi}) = \frac{-1}{\eta}\log(1-\epsilon). \tag{6.5}$$

Then, based on the expressions for $R_{1-\epsilon}(M,\overline{\phi})$, $W_{1-\epsilon}(M,\overline{\phi})$ and $W_{1-\epsilon}(G,\boldsymbol{\phi})$, we can obtain an approximation for $R_{1-\epsilon}(G,\boldsymbol{\phi})$ using

$$\frac{R_{1-\epsilon}(M,\overline{\phi})}{W_{1-\epsilon}(M,\overline{\phi})} \approx \frac{R_{1-\epsilon}(G,\boldsymbol{\phi})}{W_{1-\epsilon}(G,\boldsymbol{\phi})}$$

conjecturing that the ratio of the almost-worst-case response time to the almost-worst-case workload would be approximately equal for two queues that have the

same mean arrival rate, same mean processing speed and same job size distribution. Similar ideas have been considered in other areas of queueing theory connecting Markovian queues to general queues [Buzacott and Shanthikumar, 1993]. Essentially, we propose to estimate

$$R_{1-\epsilon}(G,\boldsymbol{\phi}) \approx R_{1-\epsilon}(M,\overline{\phi})\frac{W_{1-\epsilon}(G,\boldsymbol{\phi})}{W_{1-\epsilon}(M,\overline{\phi})}, \tag{6.6}$$

where the derivation of $W_{1-\epsilon}(G,\boldsymbol{\phi})$ is given in Equation (6.5) and $R_{1-\epsilon}(M,\overline{\phi})$ and $W_{1-\epsilon}(M,\overline{\phi})$ are given in the next section based on a mean-based approximation.

To achieve our objective of this study, i.e., searching for a minimal number of servers that guarantees the $1-\epsilon^{th}$ response times, we need to iterate through $W_{1-\epsilon}(G,\boldsymbol{\phi})$ with different number of servers, i.e., N, using our proposed approximation.

6.3 Mean-based Approximation

To obtain values of $R_{1-\epsilon}(M,\overline{\phi})$ in Equation (6.6), we propose a mean-based approximation, i.e., considering the average arrival rate, average execution speed ($\overline{\phi}$), and average number of jobs, for an $M/G/1/PS(\overline{\phi})$ queueing system. We first obtain the distribution of the number of jobs in an $M/G/1/PS(\overline{\phi})$ queueing system, conditioned upon the workload. To accommodate a high variability of job size, we approximate the job size to be degenerate hyperexponential and derive the LST expression of the number of jobs as well as response times, i.e., $W_{1-\epsilon}(M,\overline{\phi})$. Finally, combining the conditional probability of the number of jobs and $W_{1-\epsilon}$, we develop an approximation scheme for $R_{1-\epsilon}(M,\overline{\phi})$.

6.3.1 Conditional Distribution of Number of Jobs

For such a server that adopts processor sharing, let $X(t)$ be the number of jobs in the system at time t and $R_i(t)$ be the remaining amount of work to be processed for the i^{th} job in the system. The multi-dimensional stochastic process $\{(X(t),R_1(t),R_2(t),\ldots,R_{X(t)}(t)), t \geq 0\}$ is a Markov process. We denote $F_n(t,y_1,y_2,\ldots,y_n)$ as the joint probability

$$\begin{aligned} &F_n(t,y_1,y_2,\ldots,y_n) = \\ &\quad P\{X(t)=n, R_1(t)\leq y_1, R_2(t)\leq y_2,\ldots,R_n\leq y_n\}. \end{aligned}$$

Thereby the density function $f_n(t, y_1, y_2, \ldots, y_n)$ is defined as

$$f_n(t, y_1, y_2, \ldots, y_n) = \frac{\partial^n F_n(t, y_1, y_2, \ldots, y_n)}{\partial y_1 \, \partial y_2 \, \ldots \, \partial y_n}.$$

It is known that as $t \to \infty$, $f_n(t, y_1, y_2, \ldots, y_n)$ converges to the stationary distribution $f_n(y_1, y_2, \ldots, y_n)$ which is given by

$$f_n(y_1, y_2, \ldots, y_n) = (1-\rho)\frac{\lambda^n}{(N\overline{\phi})^n} \prod_{i=1}^{n} [1 - G(y_i)]$$

where $\rho = \lambda m/(N\overline{\phi})$.

Using the above we next obtain the joint probability that there are n jobs in the system and the total workload is not more than y which we denote as $\hat{F}_n(y)$ and define as

$$\begin{aligned} \hat{F}_n(y) = \lim_{t\to\infty} P\{X(t) = n, \\ R_1(t) + R_2(t) + \ldots + R_n(t) \le y\} \end{aligned}$$

for all $n \ge 0$ and $y \ge 0$. Using the expression for $f_n(y_1, y_2, \ldots, y_n)$, we have

$$\begin{aligned} \hat{F}_n(y) &= \int_0^y \int_0^{y-y_1} \int_0^{y-y_1-y_2} \ldots \int_0^{y-y_1-y_2-\ldots-y_{n-1}} \\ &\quad f_n(y_1, y_2, \ldots, y_n) dy_n dy_{n-1} \ldots dy_1 \\ &= (1-\rho)\frac{\lambda^n}{(N\overline{\phi})^n} \int_0^y (1 - G(y_1)) \\ &\quad \int_0^{y-y_1} (1 - G(y_2)) \int_0^{y-y_1-y_2} (1 - G(y_3)) \\ &\quad \ldots \int_0^{y-y_1-y_2-\ldots-y_{n-1}} (1 - G(y_n)) \\ &\quad dy_n dy_{n-1} \ldots dy_1. \end{aligned}$$

Since there is an n-folded convolution, it is only natural that we write down the LST of $\hat{F}_n(y)$, namely $\tilde{\hat{F}}_n(s)$, which after some algebra and calculus is

$$\tilde{\hat{F}}_n(s) = (1-\rho)\frac{\lambda^n}{(sN\overline{\phi})^n}[1 - \tilde{G}(s)]^n \tag{6.7}$$

for all $s \ge 0$.

It would typically be difficult to invert the above LST and obtain $\hat{F}_n(y)$ except for some special cases that we will investigate in the next subsection. Nonetheless, we go ahead and use the density $\hat{f}_n(y)$ defined as the derivative of $\hat{F}_n(y)$ with respect to y for all $y > 0$. Using that, we can write down an expression for the steady-state probability that there are n jobs in the system, given the workload, $W(t)$, is y as

$$\lim_{t\to\infty} P\{X(t) = n | W(t) = y\} = \frac{\hat{f}_n(y)}{\sum_{k=1}^{\infty} \hat{f}_k(y)}. \tag{6.8}$$

6.3.2 Special Case: Degenerate Hyperexponential Work

Here, we consider a special case of amount of work requested by jobs, i.e., degenerate hyperexponential distribution, to represent a highly varying job size as well as to demonstrate how to compute conditional probability in Equation (6.8) and tail workload.

Let H be a non-negative random variable that has a degenerate hyperexponential distribution with parameters q and θ if its CDF is $P\{H \le h\} = 1 - qe^{-\theta h}$ for some $h \ge 0$ where q satisfies $0 \le q \le 1$ and $\theta > 0$. Notice that when $q = 1$, H reduces to the standard exponential distribution, while $q = 0$ corresponds to $H = 0$, a deterministic value. Also, when $0 \le q < 1$, H has a mass at 0 with probability $1-q$. One can compute the LST of the CDF as $E[e^{-sh}] = (1-q)+q\theta/(s+\theta)$. Further, the mean and variance of H are $E[H] = q/\theta$ and $Var[H] = (2q-q^2)/\theta^2$.

We model the amount of work a job brings to the following degenerate hyperexponential distribution. We particularly consider the workload distribution that has SCOV at least 1. From a practical standpoint this would be a reasonable assumption when there are many extremely tiny jobs and the SCOV is greater than 1 (both of which are typical in many server environments). Also, from a mathematical point of view, there are just two parameters to estimate. Recall from Section 6.1 that the amount of work has a mean m, SCOV c^2 and distribution $G(\cdot)$. We can estimate q and θ by fitting the mean and SCOV. Thus we have

$$q = 2/(1 + c^2) \quad \text{and} \quad \theta = 2/(m + mc^2).$$

Distribution of Number of Jobs

Now, reverting back to Equation (6.7), we approximate the LST $\tilde{\hat{F}}_n(s)$ as

$$\tilde{\hat{F}}_n(s) = (1-\rho)\frac{\lambda^n}{(sN\overline{\phi})^n}[sq/(s+\theta)]^n$$

which can be inverted to get

$$\hat{f}_n(y) = (1-\rho)\left(\frac{\lambda q}{N\overline{\phi}}\right)^n \frac{y^{n-1}e^{-\theta y}}{(n-1)!}$$

for all $y > 0$. Thereby, we can write down the steady state probability of having n jobs at the server, given the workload is y using Equation (6.8) for all $n \geq 1$ as

$$\lim_{t\to\infty} P\{X(t) = n | W(t) = y\} = \left(\frac{\lambda q y}{N\overline{\phi}}\right)^{n-1} \frac{e^{-\lambda q y/(N\overline{\phi})}}{(n-1)!}$$

which is a Poisson distribution with parameter $\lambda q y/(N\overline{\phi})$ with the adjustment made from n to $n-1$ since n cannot be zero if $y > 0$. Notice that the number of jobs at a server, given a non-zero workload does not depend on the mean job size. However, it does depend on the SCOV of job sizes through q.

Workload Distribution

Recall that we would like to analyze an $M/G/1$ queue with PP(λ/N) arrivals, and each arrival brings a random amount of work that needs to be processed by a single server. We let $\lambda' = \lambda/N$. The amount of work for various arrivals is IID degenerate-hyperexponential with a common CDF, as described earlier. The server uses a single processing rate $\overline{\phi}$. The jobs can be served according to any work-conserving discipline, i.e., FCFS or processor sharing. If the $M/G/1$ queue is observed at an arbitrary time in steady state, then let V be the time to empty all the workload in the system. The LST of V (assuming stable) can be computed as:

$$E[e^{-sV}] = \frac{(1-\rho)s}{s - \lambda(1-\tilde{G}(-s/\overline{\phi}))}$$

where $\rho = \lambda' q/(\theta\overline{\phi})$ and $\tilde{G}(-s/\overline{\phi}) = (1-q) + q\theta/(s/\overline{\phi} + \theta)$. Rearranging the terms we get

$$E[e^{-sV}] = (1-\rho) + \rho\frac{\theta\overline{\phi} - \lambda' q}{\theta\overline{\phi} - \lambda' q + s}$$

which upon inverting we get the CDF of V as

$$P\{V \leq t\} = 1 - \rho e^{-(\theta\overline{\phi} - \lambda' q)t}$$

for all $t \geq 0$. Now, the amount of workload in steady-state W_∞ can be written as $W_\infty = V\overline{\phi}$. Hence the CDF of the amount of workload is

$$P\{W_\infty \leq x\} = 1 - \rho e^{-(\theta\overline{\phi} - \lambda' q)x/\overline{\phi}}.$$

Thus, we have $W_{1-\epsilon}(M, \overline{\phi}) = \{x : P\{W_\infty \leq x = 1 - \epsilon\}\}$

6.3.3 Tail Response Time Approximation

In the remainder of this subsection, we present an approximation scheme, based on the so-called $(1-\epsilon)$–worst case analysis. Our approximation scheme for $(1-\epsilon)^{th}$ percentile response time is motivated by Little's law and based on the idea that the tail response time depends on the tail job size, the tail number of jobs, and the average execution speed.

Approximation algorithm for $(1-\epsilon)$–worst case analysis

1. Obtain the average speed, $\overline{\phi} = \mathbf{p}\boldsymbol{\phi}$, where the terms are defined near Equation (6.1)

2. Obtain the $1-\epsilon^{th}$ percentile of the steady-state workload, $W_{1-\epsilon} = -\log(\epsilon)/\eta$, according to Equation (6.4).

3. Obtain the number of jobs in the system. Using the analysis in Section 6.3.2, we obtain the number of jobs to be a Poisson random variable with parameter $\lambda q y/(N\overline{\phi})$ where for y we use $W_{1-\epsilon}$ in step 2 above and $\overline{\phi}$ is from step 1. Then, we obtain 50^{th} percentile of number of jobs in the system, termed $J_{1-\epsilon}$, corresponding to having $W_{1-\epsilon}$ amount of workload.

4. Obtain the $(1-\epsilon)^{th}$ percentile of the amount of work brought by an arriving job in steady state, $Y_{1-\epsilon} = qG^{-1}(1-\epsilon)$, where q is the parameter in the degenerate hyperexponential distribution defined in Section 6.3.2.

5. Approximate the $(1-\epsilon)^{th}$ response time as $R_{1-\epsilon} = Y_{1-\epsilon}(1 + J_{1-\epsilon})/\overline{\phi}$.

While steps 1, 2 and 4 of the above algorithm are straightforward, it is worthwhile explaining the rationale behind steps 3 and 5. As a conservative approximation we essentially assume that the tail response time corresponds to the tail job size and tail number of jobs in the system. A crucial thing to observe is that

small changes to the number of other jobs during an arriving job's response (especially when that number of other jobs is high) would not adversely affect the response time as the large job size would overwhelm, leading to the expression in step 5 and the use of the median number as a surrogate for the number of jobs in step 3.

6.4 Experimental Results

In this section, we present an extensive set of experimental results, comparing the proposed analysis against simulation results. Our objective is to show: (1) the accuracy of the proposed workload analysis; (2) the accuracy of the response time approximations; and (3) the optimality of the dimensioned cluster size. To such an end, we consider a large number of system and workload parameters, in particular, the number of servers, different execution speeds, and job size variability. The metrics of interest are the tail workload distributions and the tail response time distributions, i.e., the 50^{th}, 75^{th}, 85^{th}, 95^{th}, 99.5^{th}, 99.95^{th}, 99.995^{th}, and 99.9995^{th} percentiles. Due to lack of space we present only partial results. In the following, we first explain the experiment settings, followed by the accuracy and scalability of the proposed analysis, and finally, the analysis of optimizing the cluster for a target tail response time.

6.4.1 Experiment Setup

We develop a simulator based on OMNet++ [OMNeT++, 2015] for the system shown in Figure 6.1, consisting of N queues, i.e., $N = \{1, 4, 10, 20, 30\}$, each of which execute requests in a processor sharing fashion. Requests following Poisson distribution with rates $\lambda = \{0.8, 3.2, 8, 16, 24\}$, corresponding to $N = \{1, 4, 10, 20, 30\}$, respectively, are dispatched to each queue by a round-robin load balancer. Each server experiences different execution speeds, governed by the $\mathbf{Q}$ matrix of CTMC. In particular, we consider two scenarios: (1) servers alternate between 2 speeds, $\boldsymbol{\phi} = [8, 10]$ according to $\mathbf{Q_2}$, (2) servers alternate between 5 speeds, $\boldsymbol{\phi} = [8, 9, 10, 11, 12]$ according to $\mathbf{Q_5}$, where $\mathbf{Q_2}$ has all non-diagonal entries $q_{ij} = 0.05$ and $\mathbf{Q_5}$ has all non-diagonal entries $q_{ij} = 0.1$.

Each job brings a workload amount with mean $m = 10$ KBytes and different SCOV, i.e., $c^2 = \{1, 4, 9\}$. The job sizes follow the degenerate hyperexponential distributions described in Section 6.3.2. Note that we purposely make the values of $\mathbf{Q_2}$ and $\mathbf{Q_5}$ comparable so as to study the effect of increasing the number of speed choices $|\boldsymbol{\phi}|$.

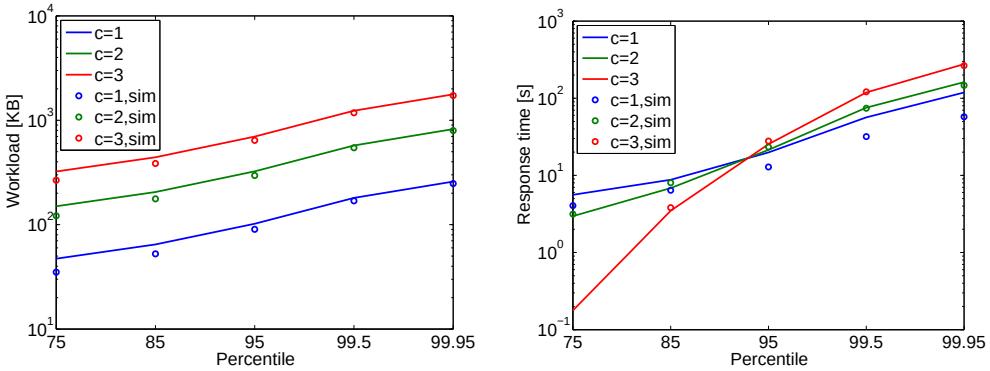
Exp	Parameters					Workload error[%]					Response Times error [%]				
	N	λ	$\|\phi\|$	c	ρ	p75	p85	p95	p99.5	p99.95	p75	p85	p95	p99.5	p99.95
1	1	0.8	2	1	0.89	9.1	6.0	2.9	0.6	0.1	88.2	101.6	138.6	181.1	213.3
2	1	0.8	2	2	0.89	8.8	6.1	3.5	1.6	1.1	7.1	13.4	14.7	24.5	32.0
3	1	0.8	2	3	0.89	8.9	6.3	3.6	1.6	3.6	n/a	4.8	6.2	4.0	0.1
4	4	3.2	2	1	0.89	18.1	12.6	7.4	3.8	2.3	34.5	46.1	62.4	94.5	113.6
5	4	3.2	2	2	0.89	11.5	8.2	4.9	2.2	0.9	0.9	5.8	2.3	8.3	13.9
6	4	3.2	2	3	0.89	10.2	7.2	4.3	1.9	1.6	n/a	5.0	11.5	9.9	5.3
7	10	8.0	2	1	0.89	29.2	21.9	15.2	10.4	8.6	27.3	42.3	62.4	86.1	105.4
8	10	8.0	2	2	0.89	13.9	10.3	6.9	4.6	3.6	8.4	4.2	5.2	4.5	11.9
9	10	8.0	2	3	0.89	11.3	8.2	5.4	3.2	2.1	n/a	4.3	13.6	10.7	5.4
10	20	16.0	2	1	0.89	17.1	10.9	5.2	1.5	0.7	16.5	30.3	40.2	65.2	84.8
11	20	16.0	2	2	0.89	10.6	7.2	4.0	1.8	1.2	10.9	6.8	7.8	1.8	9.1
12	20	16.0	2	3	0.89	10.0	7.0	4.0	1.9	1.7	n/a	5.6	14.7	11.8	6.6
13	30	24.0	2	1	0.89	20.9	14.2	8.1	3.5	2.1	19.5	21.5	43.5	68.7	83.4
14	30	24.0	2	2	0.89	11.5	8.0	4.6	2.6	2.2	10.2	6.1	7.1	2.4	7.6
15	30	24.0	2	3	0.89	10.3	7.3	4.4	2.2	2.3	n/a	5.3	14.4	11.5	7.3
16	1	0.8	5	1	0.80	15.5	9.8	4.8	1.1	1.0	60.2	76.7	116.3	153.0	183.6
17	1	0.8	5	2	0.80	18.2	12.4	7.1	3.0	3.0	6.3	3.1	7.5	16.9	25.2
18	1	0.8	5	3	0.80	18.9	12.8	7.2	3.1	0.6	n/a	9.0	4.2	3.3	10.1
19	4	3.2	5	1	0.80	34.5	22.9	13.1	6.7	4.5	38.1	36.8	54.1	77.3	107.0
20	4	3.2	5	2	0.80	23.5	16.2	9.6	5.2	3.3	5.9	14.7	8.5	1.9	10.2
21	4	3.2	5	3	0.80	21.2	14.7	8.8	4.7	2.6	n/a	8.8	8.9	1.9	4.7
22	10	8.0	5	1	0.80	53.9	36.9	23.3	14.8	11.8	17.2	20.8	45.1	75.2	99.4
23	10	8.0	5	2	0.80	27.3	19.2	12.1	7.3	5.5	20.3	13.3	7.0	3.6	7.9
24	10	8.0	5	3	0.80	22.8	16.0	9.9	5.6	3.7	n/a	8.2	8.3	1.2	5.6
25	20	16.0	5	1	0.80	36.9	22.9	11.5	4.3	1.8	7.1	10.4	32.6	48.6	73.1
26	20	16.0	5	2	0.80	23.2	15.5	8.7	4.2	2.4	22.4	15.7	9.6	4.7	4.9
27	20	16.0	5	3	0.80	20.9	14.2	8.2	4.2	3.1	n/a	9.3	9.5	2.6	4.1
28	30	24.0	5	1	0.80	43.3	27.8	15.4	7.4	4.4	9.9	13.3	36.0	52.6	77.6
29	30	24.0	5	2	0.80	24.8	16.8	9.9	5.2	4.0	21.7	15.0	8.7	3.8	6.0
30	30	24.0	5	3	0.80	21.6	14.8	8.8	4.6	2.9	n/a	9.1	9.2	5.2	4.6

Table 6.1. Prediction errors, comparing analytical results with simulation.

The exact parameter combinations and the resulting prediction errors between the proposed analysis and the simulation results are listed in Table 6.1. The prediction error is defined as the absolute difference between the prediction and simulation result, divided by the simulation result. response times of lower percentiles, such as the 50^{th} and 75^{th} percentile, are measured as low as 0 in the simulation. In these cases the prediction error is listed as n/a in Table 6.1. In most cases, our prediction overestimates the simulation results. Note that for each combination, our simulation results are averaged across 10 runs, where each queue roughly receives 6.4 million requests. The simulation time grows exponentially with the number of speeds, servers, traffic intensities, as well as, the tail statistics. The longer simulations are in the order of several hours. In contrast, the computational time of the proposed analysis is negligible.

6.4.2 Accuracy and Sensitivity Analysis

In this subsection, we first report on the accuracy of our proposed tail workload and response time analysis, using Table 6.1, and then discuss how these two metrics vary across different parameter settings, and what their implications are on system design.



(a) Workload comparisons (b) Response time comparisons
Varying c with $N = 4$, $|\boldsymbol{\phi}| = 5$ (experiments no. 13, 14, 15)

Figure 6.2. Sensitivity analysis of tail workloads and response times.

Prediction Errors

The average prediction error is around 5% for both the workload and response time in most considered cases. For workload predictions, our model works very well particularly for higher percentiles and for highly varying workloads, i.e., for higher values of c, at a given number of servers. One can observe that in the workload part of the table, i.e., its middle section, prediction errors in the lower left corner are higher than in the upper right corner. This indicates that the workload prediction suffers slightly, in particular of overestimation, when increasing the number of servers and number of execution speeds.

As for response time predictions, our proposed approximation results in slightly higher errors compared to the workload part. In particular, our estimations are highly conservative for job size distributions with low variance, i.e., $c = 1$, and for higher tail response times. The good news is that the overestimation effect is mitigated with increasing variability of job sizes, increasing numbers of servers, and increasing number of execution speeds. Overall, our proposed analysis is very accurate in predicting the 95^{th} and 99.5^{th} percentile of response times for systems with high numbers of servers experiencing high job size variability, and frequent execution speed changes due to the external environment.

Sensitivity Analysis

To see how the distribution tail of workload and responses time changes with the job variability, we select three sets of experiments and summarize them in Figure

6.2, corresponding to the cases having $N = 4$, $|\boldsymbol{\phi}| = 5$ while varying $c = \{1, 2, 3\}$. We plot both the analytical and simulation results.

In Figures 6.2 (a) and (b), one can clearly observe that the workload increases with the degree of job variability for any given percentile. The tail workload grows exponentially, shown as almost linear curves in Figure 6.2 (a), due to the log scale. As stated earlier, our workload predictions are overestimated for lower percentiles, but very accurate for higher percentiles. In terms of tail response time distributions, a different pattern with respect to the job variability can be observed. In Figure 6.2 (b), the tail distribution of response times increases differently for different job variability values. Comparing $c = 1$ and $c = 3$, the 75^{th} and 85^{th} percentile of $c = 3$ are lower than $c = 1$, whereas for the 95^{th}, 99.5^{th} and 99.95^{th} percentile, the opposite holds true. Such an observation is not too surprising, since extensive related work has pointed out that the average response times is much worse in systems with highly varying job sizes. The additional merit of our analysis is to pinpoint at which part of the tail the response times degrade drastically and impact the average performance. This further indicates that dimensioning a cluster based on (high) tail response times is very different from using the average response time. Moreover, we again stress our response time prediction as being particularly conservative for $c = 1$, i.e., overestimating, and very accurate for $c = 2, 3$.

We also compare the above results with the cases having $N = 4$, $|\boldsymbol{\phi}| = 2$. Due to lack of space, we skip the graphical presentation and only report the observations. All observations made in the first case still hold, except the exact values are slightly higher due to the lower traffic intensity, ρ. As such, the cross point among response times for different jobs size variabilities happens at slightly higher percentiles. This highlights another important factor in dimensioning cloud clusters, i.e., the traffic intensity loading the system, but this is currently out of the scope of this study.

6.4.3 Optimizing for Cloud Clusters

The objective of this subsection is to demonstrate how our prediction can be applied in optimizing a cloud cluster so that the SLAs specified by the tail response times can be achieved with the minimum number of servers, i.e., lowest cost. To such an end, we assume that the system has a job arrival rate of $\lambda = 14$ jobs per second, and the job size follows a degenerate hyperexponential distribution with $c = 3$. The server experiences 5 different execution speeds $|\boldsymbol{\phi}| = 5$ governed by a CTMC with Q_5, described at the beginning of this section. The system also considers 15 seconds as a target value, which can be used for the SLA specified

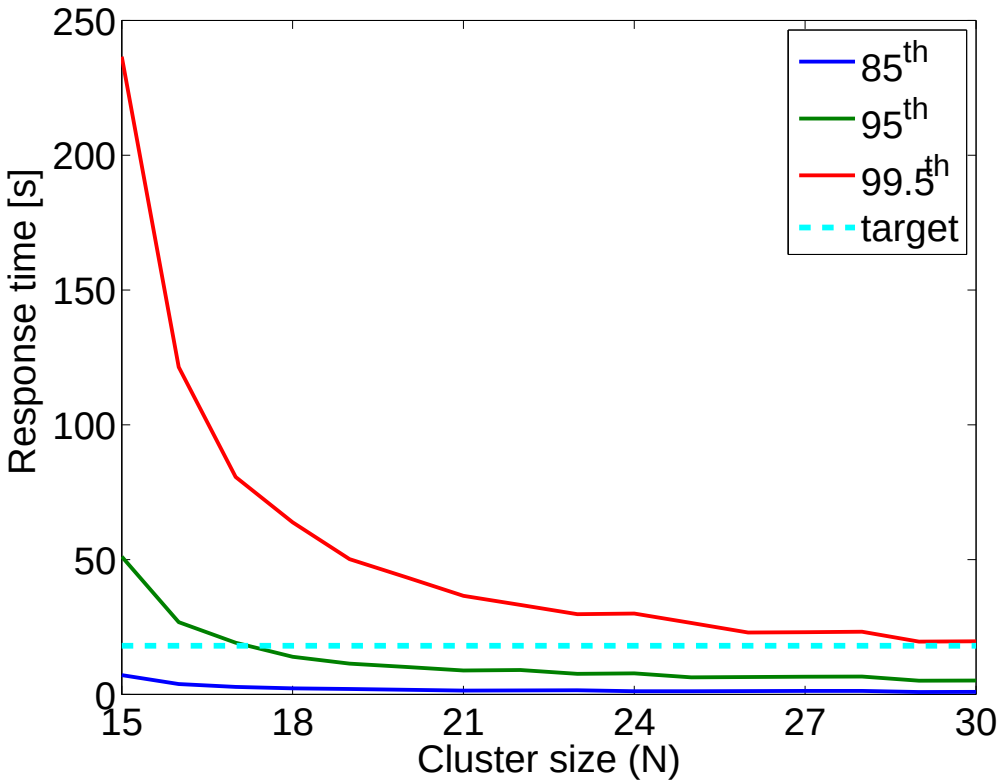


Figure 6.3. Optimal cluster size

by a tail percentile. In Figure 6.3, we show how the optimal cluster size changes across different percentiles, using 15 seconds as a target value.

In particular, we depict how the 85^{th}, 95^{th}, and 99.5^{th} percentile of response times evolve with different cluster sizes, i.e., 15-30 servers, using our analysis. On one hand, a cluster size greater than 15 achieves that the 85^{th} response time percentile is less than 15 seconds. On the other hand, 18 servers and 30 servers are the minimal cluster size to fulfill the same target at the 95^{th} and 99.5^{th} percentile, respectively. To provide statistical guarantees on the long tail of response times, a substantial service provisioning cost is unavoidable. Overall, the proposed analysis enables an efficient methodology to evaluate such emerging challenges, i.e., capturing the tail distribution of response times in highly distributed and volatile systems, i.e., in terms of workloads and server execution speeds.

6.5 Assumptions and Limitations

In this work, we make similar assumptions as in Chapters 3, 4 and 5 in terms of atomic, stateless services, but there are also some differences. In this work, we assume processor sharing for the request processing at the VMs, which is closer to real systems than the first-come-first-served policy that is assumed in the earlier chapters.

We assume that the load balancers are not able to monitor the internal state of the VMs, and therefore route jobs using round robin. Uniform random as the load balancing policy has also been investigated in the literature, but to use e.g., join-the-shortest-queue would require additional non-trivial extensions to our work.

Similarly to the work in Chapter 5, we here also focus on resource provisioning in the medium- to long-term. Therefore, we do not take into account any overhead resulting from e.g., data transfer required to be able to start up a new VM on a new physical machine.

6.6 Summary

Motivated by the emerging challenge in capturing the tail response times in cloud systems where workload and server execution speeds are highly varying, we developed an approximation of tail response times for the $G/G/1/PS$ queueing system with Markov modulated execution speeds. We first derived the tail workload distribution and then developed the approximation algorithms based on $M/G/1/PS$ queueing systems. Using extensive simulation results, we showed that our proposed analysis is particularly accurate for systems with highly varying job sizes, and large number of servers experiencing frequent execution speed changes.

Chapter 7

Conclusions

Motivated by the elasticity and ease-of-management of cloud computing, service providers seek the cloud as hosting platforms and seek for resource management solutions tailored for service workloads and cloud systems. Service providers face several challenges of provision cloud resources, arising from the service workload, cloud system, and stringent performance metrics. Service workloads are complex, e.g., composed of different atomic services, and exhibit strong time-variability. Due to the employment of virtualization technology on heterogeneous hardware, cloud systems often suffer from performance variability. Moreover, to provide competitive services to users, providers strive to guarantee not only the average performance metrics but also their higher percentiles.

This thesis develops resource provision strategies which aims to best allocate service replicas in clouds, such that multiple performance metrics, in particular the tail throughput and tail response times, can be optimally fulfilled. Applying methods in simulation and analytical modeling, policies developed in this thesis can achieve significant cost savings by effectively deploying service replicas, while adhering to target performance targets.

7.1 Contributions

To address the high problem complexity of workloads, systems, and performance metrics, this thesis considers a subset of challenges inherent to cloud provisioning, and makes specific contributions to the following aspects:

- **Delivering a two-tier application of composed services**

  We propose a replica provisioning policy which adjusts the number of service replicas in a two tier system based on the predicted workloads, such

that all replicas are well utilized at their target values. In particular, we model the workload balance and interdependency among tiers by estimating the probability that processing in the first tier replicas is not blocked waiting for work in the second tier to complete. We provide theoretical bounding analysis on first tier replicas and derive optimal/maximal nominal utilization. Our trace-driven simulation results imply great replica savings and balanced utilization of resources on both tiers, while maintaining low response times.

- **Opportunistic provisioning**

  We develop an opportunistic replication policy for elastic service provisioning on cloud platforms that optimizes the cost and performance not only for a single service, but also for the entire system. By leveraging the variability in VM performance and pay-as-you-go billing contracts in the cloud, the number of VMs for each service is minimized and opportunistically provisioned with better performing VMs. We assume a system consisting of multiple service types, where each service is stateless and atomic, and can easily be replicated to provide service scalability. Our proposed policy achieves lower cost and better performance in terms of VM utilization and response time, compared to existing replication policies that are oblivious to performance and billing characteristics of the cloud.

- **Capturing tail throughput**

  We develop a QoS-aware VM provisioning policy using a Markovian framework which explicitly models the capacity variability of a service cluster, and derives a probability distribution of QoS fulfillment. In particular, we examine atomic, stateless services of a single type. To achieve the guaranteed QoS at minimal cost, we construct theoretical and numerical cost analyses, which facilitate the search for an optimal size of a given VM configuration, and additionally support the comparison between VM configurations. Our results also give counter examples on why e.g., average-based provisioning of VMs cannot strike an optimal balance of cost and QoS fulfillment in the clouds with performance variability.

- **Capturing tail response times**

  We derive an approximation scheme to capture the response time percentiles, based on large deviation theory. The specific systems considered cater to atomic services in the cloud, essentially a G/G/1/PS queueing systems and have virtual capacities that are subject to exogenous variability.

Via simulation, the derived scheme is particularly accurate for systems hosting large number of replicas, and experiencing highly varying workloads in high intensities. Moreover, the proposed approximation can be applied to optimize the size of service clusters hosted in the cloud.

7.2 Limitations and Future Work

Despite the contributions presented in this thesis, the complex nature of service provisioning in the cloud means that we cannot claim that our work is the silver bullet for service providers. As a result, the proposed solutions might fall short in solving the entire problem and provide only a suboptimal solution. The main limitations we have identified are the lack of extensive evaluations on real systems, and the combination of all the presented work into a comprehensive framework for service provisioning.

One of the most obvious directions for future work would be to verify some of the presented results on real cloud deployment testbeds. Setting up the application VMs, load balancers and request generators could be largely done using existing tools, VM images, configuration scripts, and load generators. Instrumenting all the different components to be able to tune and measure the critical metrics such as request rates, throughput, response times and utilization would for some pieces also be straight-forward, whereas it for others could be more challenging. Nevertheless, being able to confirm the results using a few different applications would already be extremely useful.

Another related avenue would be to explore how the different proposed solutions could be combined to support more complex, but at the same time more realistic deployments. A straight-forward starting point would be to apply the control knobs of the opportunistic resource provisioning algorithm for tuning VMs on and off, reconfiguring VMs and replacing slow VMs in combination with one or more of the other provisioning approaches. For the two-tier provisioning algorithm this would involve taking the varying performance into account in the form of the average service execution time. For the provisioning algorithm based on tail throughput, the potential increase in average capacity resulting from replacing of slow VMs would manifest as changes in the high and low capacity levels over time. As the cluster size is adjusted over time using the tail response time algorithm, better performing VMs would lead to lower response times across the board when comparing clusters of similar size.

Applying the tail throughput work on the two-tier application provisioning could be done by basing the provisioning algorithm for back-end service replicas

on the tail throughput rather than the target utilization. The calculation of the non-blocking probability, and therefore the provisioning decisions for front-end replicas, depends on the estimated response time of the back-end replicas, and would not need to be changed.

One of the main challenges for applying the work on the tail response time guarantees to the other approaches, is the different system assumptions and models. In the tail response time system, we assume round robin load balancing and processor sharing job execution on the VMs, whereas we mainly consider join-the-shortest-queue and first-come-first-served in the other scenarios. Verifying the accuracy and adapting the tail response time algorithm and approximation formulas could turn out to be quite demanding.

Finally, a more practical approach for future work would be to implement tools that would enable the approaches proposed in this thesis. This would include facilitating the collection of metrics for arbitrary applications, calculating the resource provisioning accordingly, and finally performing the actual resource provisioning and dynamic resource adjustments on actual cloud platforms such as Amazon EC2 or OpenStack.

Bibliography

Alonso, G., Casati, F., Kuno, H. A. and Machiraju, V. [2004]. *Web Services - Concepts, Architectures and Applications*, Data-Centric Systems and Applications, Springer.

Apache [2014]. Apache HTTP Server.
URL: *http://httpd.apache.org/*

Arlitt, M. and Jin, T. [2000]. A Workload Characterization Study of the 1998 World Cup Web Site, *Network, IEEE* **14**(3): 30–37.

AWS Sizing Tool [2014]. CopperEgg.
URL: *http://copperegg.com/aws-sizing-tool/*

Baresi, L., Ghezzi, C. and Guinea, S. [2007]. Towards self-healing composition of services, *Contributions to Ubiquitous Computing*, Springer, pp. 27–46.

Baykal-Gursoy, M. and Duan, Z. [2006]. M/m/c queues with markov modulated service processes, *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, ACM, p. 38.

Björkqvist, M., Chen, L. Y. and Binder, W. [2012]. Opportunistic service provisioning in the cloud, *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, IEEE, pp. 237–244.

Björkqvist, M., Chen, L. Y., Vukolić, M. and Zhang, X. [2011]. Minimizing Retrieval Latency for Content Cloud, *INFOCOM, 2011 Proceedings IEEE*, IEEE, pp. 1080–1088.

Björkqvist, M., Spicuglia, S., Chen, L. and Binder, W. [2013]. QoS-Aware Service VM Provisioning in Clouds: Experiences, Models, and Cost Analysis, *Service-Oriented Computing*, Springer, pp. 69–83.

Boxma, O. J. and Kurkova, I. [2001]. The m/g/1 queue with two service speeds, *Advances in Applied Probability* pp. 520–540.

Buzacott, J. A. and Shanthikumar, J. G. [1993]. *Stochastic models of manufacturing systems*, Vol. 4, Prentice Hall Englewood Cliffs, NJ.

Cardellini, V., Colajanni, M. and Yu, P. S. [1999]. Dynamic load balancing on web-server systems, *Internet Computing, IEEE* **3**(3): 28–39.

Casale, G., Mi, N. and Smirni, E. [2008]. Bound analysis of closed queueing networks with workload burstiness, *ACM SIGMETRICS Performance Evaluation Review*, Vol. 36, ACM, pp. 13–24.

Casale, G. and Tribastone, M. [2013]. Modelling exogenous variability in cloud deployments, *ACM SIGMETRICS Performance Evaluation Review* **40**(4): 73–82.

Chauhan, M. A. and Babar, M. A. [2011]. Migrating service-oriented system to cloud computing: An experience report, *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, IEEE, pp. 404–411.

Chen, L. Y., Ansaloni, D., Smirni, E., Yokokawa, A. and Binder, W. [2012]. Achieving application-centric performance targets via consolidation on multicores: myth or reality?, *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, ACM, pp. 37–48.

Chen, Y., Das, A., Qin, W., Sivasubramaniam, A., Wang, Q. and Gautam, N. [2005]. Managing Server Energy and Operational Costs in Hosting Centers, *ACM SIGMETRICS Performance Evaluation Review*, Vol. 33, ACM, pp. 303–314.

Cherkasova, L. and Ponnekanti, S. R. [2000]. Optimizing a Şcontent-awareŤ load balancing strategy for shared web hosting service, *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings. 8th International Symposium on*, IEEE, pp. 492–499.

DaCapo [2014]. The DaCapo Benchmark Suite.
URL: *http://dacapobench.org/*

Dean, J. and Barroso, L. A. [2013]. The tail at scale, *Communications of the ACM* **56**(2): 74–80.

Dejun, J., Pierre, G. and Chi, C.-H. [2011]. Resource provisioning of web applications in heterogeneous clouds, *Proceedings of the 2nd USENIX conference on Web application development*, USENIX Association, pp. 5–5.

Dorsman, J.-P., Vlasiou, M. and Zwart, B. [2013]. Parallel queueing networks with markov-modulated service speeds in heavy traffic, *ACM SIGMETRICS Performance Evaluation Review* **41**(2): 47–49.

Dustdar, S. and Juszczyk, L. [2007]. Dynamic Replication and Synchronization of Web Services for High Availability in Mobile Ad-hoc Networks, *Service Oriented Computing and Applications* **1**(1): 19–33.

EC2 [2014]. Amazon Elastic Compute Cloud.
URL: *http://aws.amazon.com/ec2/*

Ezenwoye, O. and Sadjadi, S. M. [2008]. A proxy-based approach to enhancing the autonomic behavior in composite services, *Journal of networks* **3**(5): 42–53.

Farley, B., Juels, A., Varadarajan, V., Ristenpart, T., Bowers, K. D. and Swift, M. M. [2012]. More for your money: Exploiting performance heterogeneity in public clouds, *Proceedings of the Third ACM Symposium on Cloud Computing*, ACM, p. 20.

Fehling, C., Leymann, F. and Mietzner, R. [2010]. A framework for optimized distribution of tenants in cloud applications, *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, IEEE, pp. 252–259.

Fetai, I. and Schuldt, H. [2012]. Cost-based data consistency in a data-as-a-service cloud environment, *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, IEEE, pp. 526–533.

Gautam, N. [2012]. *Analysis of Queues: Methods and Applications*, 1st edn, CRC Press.

Gupta, V. and Harchol-Balter, M. [2009]. Self-adaptive admission control policies for resource-sharing systems, *ACM SIGMETRICS Performance Evaluation Review* **37**(1): 311–322.

Gupta, V., Sigman, K., Harchol-Balter, M. and Whitt, W. [2007]. Insensitivity for ps server farms with jsq routing, *ACM SIGMETRICS Performance Evaluation Review* **35**(2): 24–26.

Jackson, K. R., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., Wasserman, H. J. and Wright, N. J. [2010]. Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud, *Cloud*

Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, IEEE, pp. 159–168.

Jackson, K. R., Ramakrishnan, L., Runge, K. J. and Thomas, R. C. [2010]. Seeking Supernovae in the Clouds: A Performance Study, *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ACM, pp. 421–429.

Kleinrock, L. [1975]. *Queueing Systems*, Vol. I: Theory, Wiley Interscience.

Kossmann, D., Kraska, T. and Loesing, S. [2010]. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ACM, pp. 579–590.

Kraft, S., Casale, G., Krishnamurthy, D., Greer, D. and Kilpatrick, P. [2011]. Io performance prediction in consolidated virtualized environments, *ACM SIGSOFT Software Engineering Notes*, Vol. 36, ACM, pp. 295–306.

Kraska, T., Hentschel, M., Alonso, G. and Kossmann, D. [2009]. Consistency rationing in the cloud: Pay only when it matters, *Proceedings of the VLDB Endowment* **2**(1): 253–264.

Krebs, R., Momm, C. and Kounev, S. [2014]. Metrics and techniques for quantifying performance isolation in cloud environments, *Science of Computer Programming* **90**: 116–134.

Kurtz, T. G. [1970]. Solutions of ordinary differential equations as limits of pure Markov processes, *Journal of Applied Probability* **7**(1): 49–58.

Lin, M., Wierman, A., Andrew, L. L. and Thereska, E. [2013]. Dynamic right-sizing for power-proportional data centers, *IEEE/ACM Transactions on Networking (TON)* **21**(5): 1378–1391.

Mahabhashyam, S. R. and Gautam, N. [2005]. On queues with markov modulated service rates, *Queueing Systems* **51**(1-2): 89–113.

Mao, M. and Humphrey, M. [2012]. A performance study on the vm startup time in the cloud, *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, IEEE, pp. 423–430.

Massey, W. A. [2002]. The analysis of queues with time-varying rates for telecommunication models, *Telecommunication Systems* **21**: 173–204.

Mell, P. and Grance, T. [2011]. The NIST Definition of Cloud Computing, *Technical Report 800-145*, National Institute of Standards and Technology (NIST), Gaithersburg, MD.
URL: *http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf*

Meng, X., Isci, C., Kephart, J., Zhang, L., Bouillet, E. and Pendarakis, D. [2010]. Efficient Resource Provisioning in Compute Clouds via VM Multiplexing, *Proceedings of the 7th international conference on Autonomic computing*, ACM, pp. 11–20.

Mi, N., Casale, G., Cherkasova, L. and Smirni, E. [2008]. Burstiness in multi-tier applications: Symptoms, causes, and new models, *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Springer-Verlag New York, Inc., pp. 265–286.

Mietzner, R., Unger, T. and Leymann, F. [2009]. Cafe: A generic configurable customizable composite cloud application framework, *On the Move to Meaningful Internet Systems: OTM 2009*, Springer, pp. 357–364.

Moser, O., Rosenberg, F. and Dustdar, S. [2008]. Non-intrusive monitoring and service adaptation for ws-bpel, *Proceedings of the 17th international conference on World Wide Web*, ACM, pp. 815–824.

Mosincat, A. and Binder, W. [2009]. Enhancing bpel processes with self-tuning behavior, *Service-Oriented Computing and Applications (SOCA), 2009 IEEE International Conference on*, IEEE, pp. 1–8.

Nelson, R. [1995]. *Probability, stochastic processes, and queueing theory: the mathematics of computer performance modeling*, Springer.

Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L. and Zagorodnov, D. [2009]. The Eucalyptus Open-source Cloud-computing System, *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*, IEEE, pp. 124–131.

OMNeT++ [2015]. OMNeT++ Discrete Event Simulator.
URL: *http://www.omnetpp.org/*

OpenNebula [2015]. OpenNebula: Flexible Enterprise Cloud Made Simple.
URL: *http://opennebula.org*

Papazoglou, M. P., Traverso, P., Dustdar, S. and Leymann, F. [2008]. Service-Oriented Computing: A Research Roadmap, *International Journal of Cooperative Information Systems* **17**(02): 223–255.

Pautasso, C., Heinis, T. and Alonso, G. [2007]. Autonomic resource provisioning for software business processes, *Information and Software Technology* **49**(1): 65–80.

Pautasso, C., Zimmermann, O. and Leymann, F. [2008]. Restful web services vs. "big" web services: making the right architectural decision, *Proceedings of the 17th international conference on World Wide Web*, ACM, pp. 805–814.

Petrucci, V., Carrera, E. V., Loques, O., Leite, J. C. and Mosse, D. [2011]. Optimized Management of Power and Performance for Virtualized Heterogeneous Server Clusters, *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, IEEE, pp. 23–32.

Ramacher, R. and Mönch, L. [2012]. Dynamic service selection with end-to-end constrained uncertain qos attributes, *Proceedings of the 10th international conference on Service-Oriented Computing*, Springer-Verlag, pp. 237–251.

Rege, K. and Sengupta, B. [1985]. Sojourn time distribution in a multiprogrammed computer system, *AT&T technical journal* **64**(5): 1077–1090.

Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H. and Kozuch, M. A. [2012]. Heterogeneity and dynamicity of clouds at scale: Google trace analysis, *Proceedings of the Third ACM Symposium on Cloud Computing*, ACM, p. 7.

Riska, A., Sun, W., Smirni, E. and Ciardo, G. [2002]. Adaptload: effective balancing in clustered web servers under transient load conditions, *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, IEEE, pp. 104–111.

Ristenpart, T., Tromer, E., Shacham, H. and Savage, S. [2009]. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds, *Proceedings of the 16th ACM conference on Computer and communications security*, ACM, pp. 199–212.

Salas, J., Perez-Sorrosal, F., Patiño-Martínez, M. and Jiménez-Peris, R. [2006]. WS⬡Replication: A Framework for Highly Available Web Services, *Proceedings of the 15th International Conference on World Wide Web*, ACM, pp. 357–366.

Sansottera, A., Casale, G. and Cremonesi, P. [2013]. Fitting second-order acyclic marked markovian arrival processes, *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, pp. 1–12.

Sansottera, A., Zoni, D., Cremonesi, P. and Fornaciari, W. [2012]. Consolidation of multi-tier workloads with performance and reliability constraints, *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pp. 74–83.

Schad, J., Dittrich, J. and Quiané-Ruiz, J.-A. [2010]. Runtime measurements in the cloud: observing, analyzing, and reducing variance, *Proceedings of the VLDB Endowment* **3**(1-2): 460–471.

Serrano, D., Patiño-Martínez, M., Jiménez-Peris, R. and Kemme, B. [2008]. An autonomic approach for replication of internet-based services, *Reliable Distributed Systems, 2008. SRDS'08. IEEE Symposium on*, IEEE, pp. 127–136.

Singh, R., Sharma, U., Cecchet, E. and Shenoy, P. [2010]. Autonomic Mix-Aware Provisioning for Non-Stationary Data Center Workloads, *Proceedings of the 7th International Conference on Autonomic Computing*, ACM, pp. 21–30.

Spicuglia, S., Chen, L. Y. and Binder, W. [2013]. Join the best queue: Reducing performance variability in heterogeneous systems, *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, IEEE, pp. 139–146.

Stewart, C., Kelly, T. and Zhang, A. [2007]. Exploiting Nonstationarity for Performance Prediction, *ACM SIGOPS Operating Systems Review*, Vol. 41, ACM, pp. 31–44.

Toffetti, G., Gambi, A., Pezzé, M. and Pautasso, C. [2010]. *Engineering autonomic controllers for virtualized web applications*, Springer.

Trummer, I., Leymann, F., Mietzner, R. and Binder, W. [2010]. Cost-optimal outsourcing of applications into the clouds, *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, IEEE, pp. 135–142.

Tsakalozos, K., Roussopoulos, M. and Delis, A. [2011]. Vm: placement in non-homogeneous iaas-clouds, *Proceedings of the 9th international conference on Service-Oriented Computing*, Springer-Verlag, pp. 172–187.

Ueda, Y. and Nakatani, T. [2010]. Performance variations of two open-source cloud platforms, *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, IEEE, pp. 1–10.

van Der Aalst, W. M., Ter Hofstede, A. H., Kiepuszewski, B. and Barros, A. P. [2003]. Workflow Patterns, *Distributed and parallel databases* **14**(1): 5–51.

Verma, A., Sharma, U., Jain, R. and Dasgupta, K. [2007]. Compass: Cost of Migration-aware Placement in Storage Systems, *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*, IEEE, pp. 50–59.

Waldspurger, C. A. and Weihl, W. E. [1994]. Lottery scheduling: Flexible proportional-share resource management, *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, USENIX Association, p. 1.

Whitt, W. [1986]. Deciding which queue to join: Some counterexamples, *Operations research* **34**(1): 55–62.

Wikipedia [2014]. Wikipedia: The Free Encyclopedia.
URL: *http://www.wikipedia.org/*

Xu, Y., Musgrave, Z., Noble, B. and Bailey, M. [2013]. Bobtail: avoiding long tails in the cloud, *Proc. NSDI*.

Ye, Z., Bouguettaya, A. and Zhou, X. [2012]. Qos-aware cloud service composition based on economic models, *Proceedings of the 10th international conference on Service-Oriented Computing*, Springer-Verlag, pp. 111–126.

You, K., Qian, Z., Tang, B., Lu, S. and Chen, D. [2009]. Qos-aware replication in service composition., *Int. J. Software and Informatics* **3**(4): 465–482.

Zhang, B. and Zwart, B. [2012]. Fluid models for many-server markovian queues in a changing environment, *Operations Research Letters* **40**(6): 573–577.

Zhang, C., Chang, R. N., Perng, C.-S., So, E., Tang, C. and Tao, T. [2007]. Qos-aware optimization of composite-service fulfillment policy, *Services Computing, 2007. SCC 2007. IEEE International Conference on*, IEEE, pp. 11–19.

Zhang, J. and Zwart, B. [2008]. Steady state approximations of limited processor sharing queues in heavy traffic, *Queueing Systems* **60**(3-4): 227–246.

Zhang, Q., Cherkasova, L., Mi, N. and Smirni, E. [2008]. A Regression-Based Analytic Model for Capacity Planning of Multi-Tier Applications, *Cluster Computing* **11**(3): 197–211.

Zhang, Q., Riska, A., Sun, W., Smirni, E. and Ciardo, G. [2005]. Workload-aware load balancing for clustered web servers, *Parallel and Distributed Systems, IEEE Transactions on* **16**(3): 219–233.

Zheng, H., Yang, J., Zhao, W. and Bouguettaya, A. [2011]. Qos analysis for web service compositions based on probabilistic qos, *Proceedings of the 9th international conference on Service-Oriented Computing*, Springer-Verlag, pp. 47–61.

Zheng, Z. and Lyu, M. R. [2008]. A Distributed Replication Strategy Evaluation and Selection Framework for Fault Tolerant Web Services, *Web Services, 2008. ICWS'08. IEEE International Conference on*, IEEE, pp. 145–152.

Zheng, Z. and Lyu, M. R. [2009]. A QoS-Aware Fault Tolerant Middleware for Dependable Service Composition, *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, IEEE, pp. 239–248.

Zhou, Y.-P. and Gans, N. [1999]. A Single-Server Queue with Markov Modulated Service Times, *Technical report*, Wharton School Center for Financial Institutions, University of Pennsylvania.